Machina.NET: A Library for Programming and Real-Time Control of Industrial Robots

Journal of
open research software

García del Castillo y López, J L 2019 Machina.NET: A Library for Programming and Real-Time Control of Industrial Robots. *Journal of Open Research Software*, 7: 27. DOI: https://doi.org/10.5334/jors.247

SOFTWARE METAPAPER

Machina.NET: A Library for Programming and Real-Time Control of Industrial Robots

Jose Luis García del Castillo y López[1,2]

[1] Harvard University Graduate School of Design, US

[2] Autodesk Inc., US

personal@garciadelcastillo.es

Machina is a .NET library for programming and control of industrial robots. It is designed to build applications that interface with robotic devices in real time. The library features a high-level API of simple, device-agnostic action verbs to issue motion requests to robots, and translates them to device-specific instructions using low-level communication protocols and managing priority queues. It also features a set of execution-related events to notify users of changes in the asynchronous state of the robot, fostering programming styles that are reactive rather than prescriptive. These features promote an enactive approach to robotics, and provide an immediate and intuitive entry point to real-time robot control, making Machina particularly suitable for controlling systems that require concurrent responsiveness to sensory or user input. While Machina currently supports mostly six-axis industrial robotic arms, it can be easily extended to any actuable device that moves in three-dimensional space, such as 3D printers, CNC machines, drones, robotic toys, etc.

Machina is geared towards users in the creative fields, like designers, artists, makers and creative coders, and promotes features such as interactivity, intuitiveness, feedback, concurrency and cross-platform compatibility, over performance or feature-fullness. We hope this framework will help ease access for novice users to the field of robotics.

Keywords: Robotics; Real-time; Interactive; Computation; Design; Art; Architecture; Digital Fabrication; C#; HRI; HMI; HCI

(1) Overview

Introduction

During the last half of the 20th Century, and beginning of the 21st, robots have permeated into many aspects of our daily lives. Any modern product that we may consume today has been manipulated by automated machines in some kind of manufacturing process: cars, furniture, food, electronics... virtually all objects that surround us have been touched by robots. More recently, we are witnessing an increasing presence of these machines outside industry, in our domestic environments: from automated vacuum cleaners in our homes to assistive devices for impaired individuals, and very soon, delivery drones and autonomous vehicles. Robots are following a similar path to that of computers: from large and expensive machines only accessible to researchers and industry, to smaller personal versions to assist us in our daily duties. But just like it happened with desktop computers, if robots are to truly permeate into our daily lives, their programming and control should be accessible to not just the highly trained, but rather to the general user.

But first of all, what is a *robot*? This question is usually the subject of endless philosophical debate, ranging from simple definitions such as "a machine capable of automatically carrying out a complex series of movements, especially one which is programmable [1]," to increasingly ontological queries on their nature: "a robot is a constructed system that displays both physical and mental agency, but is not alive in the biological sense [2]." In any case, most authors agree that a robot is essentially constituted of two parts: a physical manifestation that actuates and/or senses its environment, and a logic system that drives its behavior. And while the former, hardware design, involves a whole set of challenges on its own, the project described in this paper tries to address the challenges present in the latter: the creation of the software mechanisms to program and control robotic devices.

One of the first challenges in this field is the significant lack of universality in robot control systems. In industry, robot manufacturers usually provide highly proprietary software environments that require the use of custom programming languages, and it is fairly common for vendors to offer their own authored tools for programming and simulating their products. For example, ABB [3] offers their licensed RobotStudio [4] platform to program robot cells in the RAPID language [5]; KUKA [6] provides KUKA.Sim [7] to simulate programs written in Kuka Robot Language [8]; and Universal

Robots [9] uses the URScript [10] language to drive six-axis robotic arms through URSim [11]. Different robot brands requiring different programming languages is the first entry barrier to robotics, tying the initial learning process to the products of a particular company. However, the kinematic similarities between six-axis robotic arms, and the universal nature of spatial dynamics, make it possible to abstract many of the principles behind motion planning into higher-level interfaces, which can then be post-processed into device-specific languages. This is the core principle behind third-party robot programming environments such as RoboDK [12] or Visual Components [13]. But in any case, all these tools are still closed source tools, with usage often restricted to the acquisition of expensive licenses, becoming an additional entry barrier for novice users to the field of robotics.

An additional problem with these industrial robot programming tools is that they often require significant domain expertise from the user. They are designed for engineers, integrators and implementers, and assume their users are familiar with notions such as forward and inverse kinematics, spatial transformations, robot mechanics and computer programming. The Robot Operating System (ROS) [14] is notorious for the depth of possibilities it allows, its modularity, openness and extensibility, but also for its complexity, steep learning curve and difficulty to set up, resulting in a reduced community of power users formed by academics and research labs. There is an abundance of highly specialized robot programming tools for highly skilled individuals, although without the proper training and time investment, robotics are hardly accessible to more general audiences.

The recent democratization of digital fabrication tools has sparked renewed interest in the field of robotics from more creative communities. The increasing availability of 3D printers, laser cutters or Computer Numeric Control (CNC) routers has given designers, architects and artists tools for rapid prototyping, and with the growth of Fab Labs [15] and maker culture [16], larger communities are becoming empowered with the production means to *make almost anything* [15]. The Arduino open-source platform [17] stands as a great example of a system that integrates a physical sensing/actuating device with a simple and intuitive development environment, designed to introduce novice makers to electronic prototyping. Personal fabrication [18] projects such as *RoMA* [19] demonstrate the potential of robot integration in ad hoc design-while-making systems. Robotic fabrication in architecture has also seen a tremendous development in the last decade, with many architecture schools incorporating robotics as part of their curricula, and dedicated conferences promoting research in the field [20–23]. Projects such as Flowing Matter [24] and the CEVISAMA pavilions [25] explore the promising possibilities that design to robotic fabrication workflows may open up for innovative building structures. Additionally, art projects such as Mimic [26] and Mimus [27] demonstrate the expressive potential that six-axis robotic arms have beyond their mere utilitarian implementations.

While industrial robotic arms are commonly used in these kinds of projects, they are seldom implemented or controlled via the software tools that their vendors provide. There are multiple reasons for this: in addition to the complexity and steep learning curves already discussed, these tools offer poor integration with common Computer Aided Design (CAD) and Manufacturing (CAM) environments typically used by these audiences. The recent popularity of visual programming languages for CAD like Grasshopper3D [28] led to a bloom of robot-oriented plug-ins, such as HAL [29], KukaPRC [30], Taco ABB [31] or Scorpion [32], which allow for the compilation of robot programs in native languages directly from the CAD environment. And while these tools offer a smoother learning curve and easier CAD to CAM workflows, they are still in many cases vendor specific and closed source, making it very hard to extend or customize for novice uses.

Moreover, industrial robots, and the environments designed to control them, are heavily biased towards the *offline programming* model. In this scenario, all motion planning, logic and instructions are pre-defined inside the programming environment, become post-processed into a program in the device's native language, separately loaded to its controller and executed offline, without connection to the tool from which it was created; this is for instance the typical use case for any common 3D printing operation. Such paradigm is well suited for highly calibrated and predictable environments, where precise assumptions about their state can be done a priori. However, it is particularly inadequate for systems with elevated degrees of uncertainty, and where robot behavior is driven reactively by mutating conditions around it. This is particularly the case of robotic systems based on sensory input, environmental feedback, material properties, interactive installations, human-robot collaboration, etc. Projects willing to explore these features must often develop their own ad hoc robot control tools, usually relying on tricks and hacks, and thus requiring significant programming experience and robot control knowledge–yet another barrier to innovative robotic exploration.

From an engineering perspective, we have gotten quite good at making machines that are precise, reliable, and fast, and that allow us to make further things with them. However, for non-power users, they can be notoriously hard to use and extend [33]. Therefore, while further research will continue to improve the hardware side of industrial robots, it could be argued that the big challenge for their mass adoption remains the software tools available to program them. For industrial robots to permeate further into non-industrial contexts, the tools used to control them should also address the qualities and needs of the new people who will use them. Training the general public in the skills of industrial robots is unrealistic, and for further progress to be made in fields such as design robotics [34] or personal fabrication [35], the next frontier of research should be the design of appropriate robotic software interfaces catered to their needs.

This paper presents ***Machina***, a .NET library for programming and real-time control of industrial robots. It features a high-level, unitary, simple and human-relatable Application Programming Interface (API) of actions to describe spatial motion, which can then be post-processed

into a program in the device's native language. Its design follows the principles of the *Enactive Robotics* model [48], providing a more suitable cognitive model for beginners, while providing a deeper spectrum of possibilities to advanced users. Even though it can be used for classical offline programming, the main emphasis of the project is real-time communication and control of robotic devices. The library defines classes with protocols for hardware drivers, low-level communication, instruction buffering, and machine state representations, which can then be extended to the particularities of any specific machine. An additional API of asynchronous events tied to execution tracking allows for programming styles that are reactive to concurrent execution by the machine at run time. While the main focus of the library are six-axis industrial robotic arms, it can (and hopefully will) be easily extended to drive any mechanical device that can sense or be actuated in three-dimensional space, such as 3D printers, CNC machines, Arduino boards, mobile robots, drones, etc.

The target audience of this project are individuals without engineering backgrounds who want to use robots for creative projects outside industry, such as designers, artists, architects, makers and creative coders. The library is designed to be immediate and intuitive to use, foster interactivity, provide constant feedback and allow concurrency, with less emphasis on other qualities such as performance or feature-fullness. The goal of this effort is to provide an entry point for non-experts to the field of robotics, open it up to research speculation, experimentation and creative inquiry, and make robotics more accessible to wider audiences.

Implementation and architecture

Hello Robot

A simple application in C# built with Machina could look like Code Sample 1:

```
using System;
using Machina;

namespace Sample
{
    class MachinaSample
    {
        static void Main(string[] args)
        {
            Robot arm = Robot.Create("HelloRobot",
            "ABB");
            arm.ControlMode("online");
            arm.Connect("192.168.125.1", 7000);

            arm.Message("Hello Robot!");
            arm.SpeedTo(100);
            arm.MoveTo(400, 300, 500);
            arm.Rotate(0, 1, 0, -90);
            arm.Move(0, 0, 250);
            arm.Wait(2000);
            arm.AxesTo(0, 0, 0, 0, 90, 0);

            Console.WriteLine("Press any key to
            finish this program");
            Console.ReadKey();

            arm.Disconnect();
        }
    }
}
```

Code Sample 1: A simple `Hello Robot` program.

In this example, the sequence of actions is as follows:

- Define a new instance of a `Robot` object, giving it a name, and defining its brand.
- Connect to a physical robot at given `IP` and `Port` values.
- Display the `"Hello Robot!"` message on the device's display.
- Set the internal linear speed value to `100` mm/s.
- Move to Cartesian XYZ coordinates `(400, 300, 500)` in the robot's reference frame.
- Rotate `-90` degrees around the unitary positive Y vector `(0, 1, 0)`.
- Move `250` mm in positive Z.
- Wait for `2000` milliseconds.
- Set the rotational values of the robot's six axes to `(0, 0, 0, 0, 90, 0)` degrees respectively.
- Halt program execution by requesting user input, giving time to the robot to complete these actions.
- Disconnect from the controller before leaving the program.

The result of the execution of this program on an ABB IRB 1200 robot is illustrated in **Figure 1**.

Syntax

Machina is designed to program machines that perform motion in three-dimensional space. It features a high-level API of instructions to describe kinematic transformations, composed of simple English verbs that denote calls to action. Its interaction model is highly influenced by the cognitive principles behind the Logo programming language [36] and its popular application to turtle graphics. Some of its syntactical flavor and state-based settings are influenced by the Processing programming language [37].

The `Robot` class is the main public interface with the library. It features methods such as `Move`, `Rotate` and `Transform` to request kinematic transformations, `Speed` and `Precision` to request changes in motion properties, or `Connect` and `ControlMode` as management methods. A program can be created by the simple concatenation of a sequence of actions, and running it will result in either the buffered execution of those actions on the connected device, or the offline compilation of that same program into the device's native language.

Where applicable, most of the API methods have two syntactical flavors: the function name with the `To` suffix, and the plain version. This denotes the difference between *absolute* and *relative* action requests respectively. Relative actions build on top of the current state of the robot, while absolute actions set those values independently of former states. For example, a `MoveTo(400, 300, 500)` call will set the position of the robot's Tool Center Point (TCP) to XYZ `(400, 300, 500)` mm, regardless of its previous position. If followed by a relative `Move(0, 0, 250)` call, the resulting absolute position of the robot would be `(400, 300, 750)`. This applies to TCP location, orientation, and robot axes values, but also to internal robot motion properties such as speed, acceleration, precision, analog out values, etc.

A breakdown of the most relevant methods in the `Robot` class can be found in **Tables 1** to **3**.

Figure 1: The four main motion actions of the `Hello Robot` program (Code Sample 1) executed on an ABB IRB 1200 robot – `MoveTo(400, 300, 500)`, `Rotate(0, 1, 0, -90)`, `Move(0, 0, 250)` and `AxesTo(0, 0, 0, 0, 90, 0)`.

Table 1: Main motion control actions in Machina.

Instruction	Description	Related Action
`Move/To`	Change the position of the TCP maintaining its orientation.	`ActionTranslation`
`Rotate/To`	Change the orientation of the TCP maintaining it position.	`ActionRotation`
`Transform/To`	Change the position and orientation of the TCP.	`ActionTransformation`
`Axes/To`	Change the rotational values of the robot's axes.	`ActionAxes`
`ExternalAxis/To`	Change the values of one the external axis attached to the robot.	`ActionExternalAxis`
`Wait`	Halt program execution for an amount of time.	`ActionWait`
`DefineTool`	Define `Tool` properties on the robot.	`ActionDefineTool`
`Attach`	Attach a `Tool` to the robot's flange.	`ActionAttach`
`Detach`	Detach all tools from the robot.	`ActionDetach`
`WriteDigital`	Writes a value to a digital out.	`ActionIODigital`
`WriteAnalog`	Writes a value to an analog out.	`ActionIOAnalog`
`Extrude`	Turns extrusion on/off in 3D printers.	`ActionExtrude`
`Message`	Display a message on the device's screen.	`ActionMessage`
`Comment`	Insert a custom comment on a compiled program.	`ActionComment`
`CustomCode`	Insert a custom line of code on a compiled program.	`ActionCustomCode`
`Do`	Applies an `Action` object to the `Robot`.	`N/A`

Table 2: Main settings actions in Machina. Note that all state changes caused by these actions can be buffered and reverted with the use of `PushSettings` and `PopSettings`.

Instruction	Description	Related Action
`MotionMode`	Set the motion type for future motion Actions, like `linear` or `joint`.	`ActionMotion`
`Speed/To`	Change the TCP speed value new Actions will be executed at.	`ActionSpeed`
`Acceleration/To`	Change the TCP acceleration value new Actions will be executed at.	`ActionAcceleration`
`Precision/To`	Change the TCP smoothing radius value new Actions will be executed at.	`ActionPrecision`
`Temperature/To`	Change the working temperature of one of the device's parts.	`ActionTemperature`
`ExtrusionRate/To`	Change the extrusion rate of filament for 3D printers.	`ActionExtrusionRate`
`PushSettings`	Buffers current state settings.	`ActionPushPop`
`PopSettings`	Reverts the settings to the previous state buffered by `PushSettings`.	`ActionPushPop`

Table 3: Main management methods in Machina. These instructions have no actions associated to them, as they are related to setup rather than execution.

Instruction	Description	Related Action
`Robot.Create`	Create a new instance of a `Robot` object.	N/A
`ControlMode`	Sets the control type the robot will operate under, like `offline` or `online`.	N/A
`ConnectionManager`	Defines who is responsible for setting up the controller for connection, Machina or the `user`.	N/A
`Connect`	Connects to a remote controller.	N/A
`Compile`	Create a program in the device's native language with all the buffered Actions.	N/A

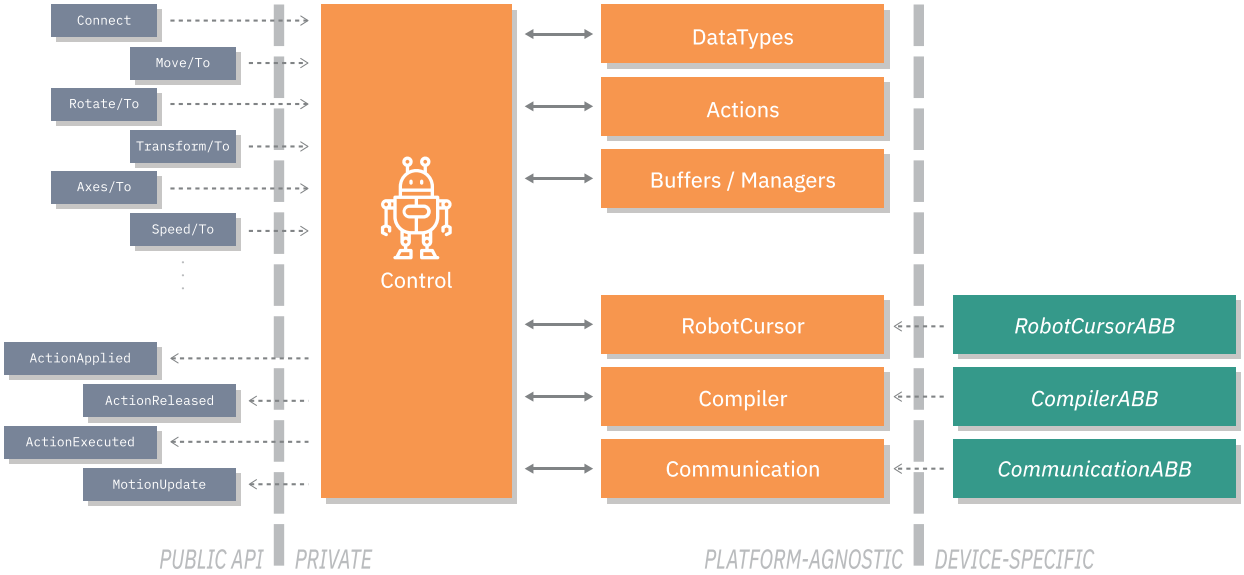

Figure 2: Overall architecture of the Machina.NET library and an extension sample for ABB robots.

Action Model

The internal programming and control model of Machina is based on the atomic form of `Action` objects. Actions are the basic units of interaction with the machine, and follow the principles of the *Enactive Robotics* model for concurrent machine control [48]. An `Action` is defined as a change in some of the properties that define the state of a robot. For example, this can mean to perform some kind of spatial motion (`Move`, `Rotate`), change the execution properties of forthcoming actions (`Speed`, `MotionMode`), hold execution for a certain amount of time (`Wait`), change the IO values (`WriteDigital`, `WriteAnalog`), etc.

Tables 1 and **2** denote a loose correlation between `Robot` methods and `Action` types. This is not coincidental, as most API calls are basically thin wrappers that create an `Action` object internally and issue a request to the core `Control` class to execute that action **Figure 2**. This pattern presents a clear mental model to the user, who can easily relate the statement `arm.Move(0, 0, 250)` to the meaning "ask the robot to move up 250 mm," and doesn't need to care about the internal representation of such actions. It also helps maintain the modularity of the library and simplifies the extension of its functionality.

Actions are platform-agnostic. They are objects which store values for the state properties they are meant to modify, but do not need information on their target machine in order to exist–even if they are created through the `Robot` class. This means that programs are created internally as lists of `Action` objects, but that these actions have no real meaning until they are applied to a particular device. Depending on its nature, such a device may or may not be able to accommodate those changes; it is the responsibility of the developer who extends the library for a new machine type to decide which actions have effects on it and what their effects are. This means that calling the `bot.ExtrusionRateTo(100)` method on a non 3D printer is a valid operation, but will have no effect on the machine when executed online or compiled offline, beyond a warning message. This allows to maintain the cross-compatibility of the same API between different types of robots. However, it is important to note that the philosophy of the library is to provide a high-level programming language as constant in behavior as possible across different machine types. While developers are welcome to contribute to the project with extensions for new machine types, only those who maintain the library's consistency will be accepted to the main project.

Machina can be extended with new actions by inheriting from the main `Action` class, and adding corresponding methods to the `Robot` class. Child actions must implement the `ToString()` method with a human-readable representation of the effect of the action; this helps the user understand the abstractions behind them. Child actions must also implement the `ToInstruction()` method to return a string mirroring the necessary API call to create such `Action`. This is extremely useful as a form of serialization, and provides a way to marshal actions when transferring programs between different implementations of the Machina framework, either through different programming languages or communication protocols. **Code Samples 2** and **3** show examples of this behavior for the action in our `Hello Robot` program (**Code Sample 1**).

Control Modes

Machina can be used for *offline* programming. In this scenario, the library works by buffering all actions into a program, which can then be compiled into the device's native language, manually loaded and ran on it. This is very similar to the typical way Arduino boards are programmed [17].

Code Sample 4 shows the `Hello Robot` example rewritten to work in `offline` mode. In this case, `program` contains a string representation of the Machina actions post-processed into ABB's native RAPID language (**Code Sample 5**). The same program, compiled in Universal Robots' URScript language is shown on **Code Sample 6**.

Note how, by default, Machina inserts comments throughout the program with human-readable descriptions of the code. This helps novice users understand the correlation between native instructions and the original Machina actions, facilitating debugging and serving as an entry point to native robot programming. This behavior can be customized through the overloads in `Compile()`.

When working in *online* mode, Machina is intended to be used in control applications running on a host on the same network as the controlled device. In this scenario, the device's controller must be running an instance of a custom *Machina driver module* written in the device's language. Depending on the device, the driver may automatically be deployed by the application, or should be manually set up by the user. The driver module allows the Machina application to exchange information with the device via whichever communication protocol the device accepts. Following the Arduino analogy, this would be akin to interfacing with the board via the Firmata protocol [38].

Figure 3 describes a high-level representation of a sample communication scheme between a Machina host application and the driver. In this example:

- Communication is established between host and driver through the device's available communication protocol.
- A handshake operation is performed to verify version compatibility and to update the host with initial machine state.
- Host streams action requests to the driver.
- Driver sends acknowledgement messages on action completion.
- Driver streams motion update messages when available.

In the example, communication with ABB devices works by running a driver module that creates a TCP server on the device, accepting incoming socket connections. On successful connection, the client receives string state messages in the format `>21 X Y Z QW QX QY QZ;` and `>22 J1 J2 J3 J4 J5 J6;` with the values for the current Cartesian and joint poses respectively. A request to transform linearly to a Cartesian pose can be sent as `@ID 1 X Y Z QW QX QY QZ;`, and upon completion of this request, the driver will send an acknowledgment message `@ID 1;` including the id value of the completed action. If capable of, the driver may also send state messages periodically, to keep the client updated on the motion state of the device.

However, Machina doesn't impose a particular specification on how drivers for new robots should be implemented. Due to the lack of universality in firmware applications, and the variability of technologies available on back-end interfaces, different devices may require different communication protocols, connection schemes, and/or message formatting. Developers willing to extend Machina for new machines are welcome to choose how to develop their own driver modules, and which technology

Code Sample 2: Actions in the `Hello Robot` example stringified to a human-readable program.

```
Display message "Hello Robot!"
Set TCP speed to 100 mm/s
Move to [400, 300, 500] mm
Rotate -90 deg around [0, 1, 0]
Move [0, 0, 250] mm
Wait 2000 ms
Set joint rotations to [0, 0, 0, 0, 90, 0] deg
```

Code Sample 3: Actions in the `Hello Robot` example serialized into instruction calls.

```
Message("Hello Robot!")
SpeedTo(100)
MoveTo(400, 300, 500)
Rotate(0, 1, 0, -90)
Move(0, 0, 250)
Wait(2000)
AxesTo(0, 0, 0, 0, 90, 0)
```

Code Sample 4: The `Hello Robot` example rewritten for `offline` mode.

```
Robot arm = Robot.Create("HelloRobot", "ABB");

arm.ControlMode("offline");

arm.Message("Hello Robot!");
arm.SpeedTo(100);
arm.MoveTo(400, 300, 500);
arm.Rotate(0, 1, 0, -90);
arm.Move(0, 0, 250);
arm.Wait(2000);
arm.AxesTo(0, 0, 0, 0, 90, 0);

List<string> program = arm.Compile();
```

Code Sample 5: The `Hello Robot` program compiled to RAPID language by the code in Code Sample 4.

```
MODULE HelloRobot_Program
    CONST speeddata vel20 := [20,20,5000,1000];
    CONST speeddata vel100 := [100,20,5000,1000];
    PROC main()
        ConfJ \Off;
        ConfL \Off;
        TPWrite "Hello Robot!"; ! [Display message "Hello Robot!"]
        ! [Set TCP speed to 100 mm/s]
        MoveL [[400, 300, 500], [0, 0, 1, 0], [0,0,0,0], [9E9,9E9,9E9,9E9,9E9,9E9]], vel100, z5,
Tool0\WObj:=WObj0; ! [Move to [400, 300, 500] mm]
        MoveL [[400, 300, 500], [0.7071, 0, 0.7071, 0], [0,0,0,0], [9E9,9E9,9E9,9E9,9E9,9E9]],
vel100, z5, Tool0\WObj:=WObj0; ! [Rotate -90 deg around [0, 1, 0]]
        MoveL [[400, 300, 750], [0.7071, 0, 0.7071, 0], [0,0,0,0], [9E9,9E9,9E9,9E9,9E9,9E9]],
vel100, z5, Tool0\WObj:=WObj0; ! [Move [0, 0, 250] mm]
        WaitTime 2; ! [Wait 2000 ms]
        MoveAbsJ [[0, 0, 0, 0, 90, 0], [9E9,9E9,9E9,9E9,9E9,9E9]], vel100, z5, Tool0\WObj:=WObj0;
! [Set joint rotations to [0, 0, 0, 0, 90, 0] deg]
    ENDPROC
ENDMODULE
```

Code Sample 6: The same `Hello Robot` program in Code Sample 4 compiled to URScript.

```
def HelloRobot_Program():
    popup("Hello Robot!", title="Machina Message", warning=False, error=False) # [Display message
"Hello Robot!"]
    # [Set TCP speed to 100 mm/s]
    movel(p[0.4,0.3,0.5,0,3.141593,0], a=0.2, v=0.1, r=0.005) # [Move to [400, 300, 500] mm]
    movel(p[0.4,0.3,0.5,0,1.570796,0], a=0.2, v=0.1, r=0.005) # [Rotate -90 deg around [0, 1, 0]]
    movel(p[0.4,0.3,0.75,0,1.570796,0], a=0.2, v=0.1, r=0.005) # [Move [0, 0, 250] mm]
    sleep(2) # [Wait 2000 ms]
    movej([0,0,0,0,1.570796,0], a=8.726646, v=0.698132, r=0.005) # [Set joint rotations to [0, 0,
0, 0, 90, 0] deg]
end
```

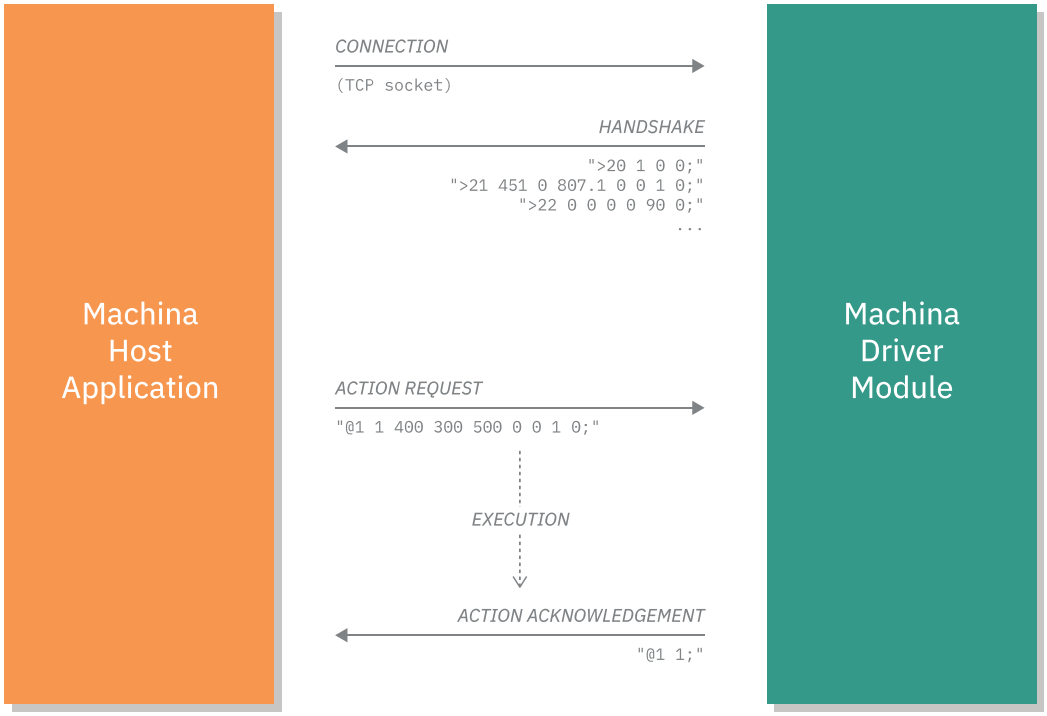

Figure 3: Sample scheme of a Machina application-driver communication exchange.

stack to implement in order to interface with them. The only requirement is to maintain the updating logic of the robot cursors that represent the asynchronous state of the system (see State Model section).

State Model

When using software applications to control hardware devices, there is a tremendous mismatch between the execution time frames of both parties: a large program

may execute in a handful of milliseconds on the digital side, but may require several minutes, if not hours, to complete on its analog counterpart. For this reason, there is an inherent challenge in coordinating the elements of a system whose parts work at such different time scales.

Machina approaches this problem by implementing a system of layered machine state representations named *cursors*. A `Cursor` is a virtual representation of the *state* of the robot, defined as the values of all possible properties a particular device may exhibit at a certain stage of the robot's execution timeline. These may include position, orientation, IO values, temperature, etc., properties which are particular to the device's capabilities. A simple extension of the `Cursor` object for a six-axis robotic arm could implement properties such as `Position` vector, an `Orientation` quaternion or an array of angular values for the `Axes`, whereas a cursor representing a conventional 3D printer may implement `Position` and `Temperature`, but not require `Orientation`.

Since `Actions` are platform-agnostic, their effect depends on the state of the device at the time of execution. An `Action` is said to be *applied* to a `Cursor` when the cursor representing that state changes its properties based on the nature of the `Action`. An example of this would be the `Position` of the cursor changing from `(400, 300, 500)` to `(400, 300, 750)` after a `Move(0, 0, 250)` action has been applied to it. This model is particularly helpful when controlling machines that do not natively accept motion instructions in relative form, since the absolute values for low-level instructions are always available application-side through the `Cursor` representation of the robot. It is also useful when switching between actions defined in Cartesian and joint coordinates, as the state can be maintained in parallel on both spaces, with translations between them updated through forward and inverse kinematic equations.

The asynchronous differences between the host application run time and the robot operation timeline are tracked through a set of cursors representing the different stages of program execution. To better understand this architecture, it will be useful to break down the different stages in the execution cycle of an `Action` **Figure 4**:

- An `Action` is *issued* when a request to execute that action has been invoked. This happens typically on most `Robot` API methods such as `MoveTo(400, 300, 500)`. Issued actions are immediately applied to the `IssueCursor`, which therefore maintains a representation of the future state of the robot after program completion, and is the base state on top of which new issued actions are applied.
- Upon issue, Machina queues actions into a first in, first out buffer, an manages the queue according to control mode and robot execution. An `Action` is *released* when it leaves the buffer, and a request to execute that action is sent to the controller. In offline mode, all actions are released at once upon `Compile()`. In online mode, Machina stages the release of actions to the controller in discrete blocks based on the amount of pending actions on it. Depending on the characteristics of the device, its microcontroller may not have enough resources to

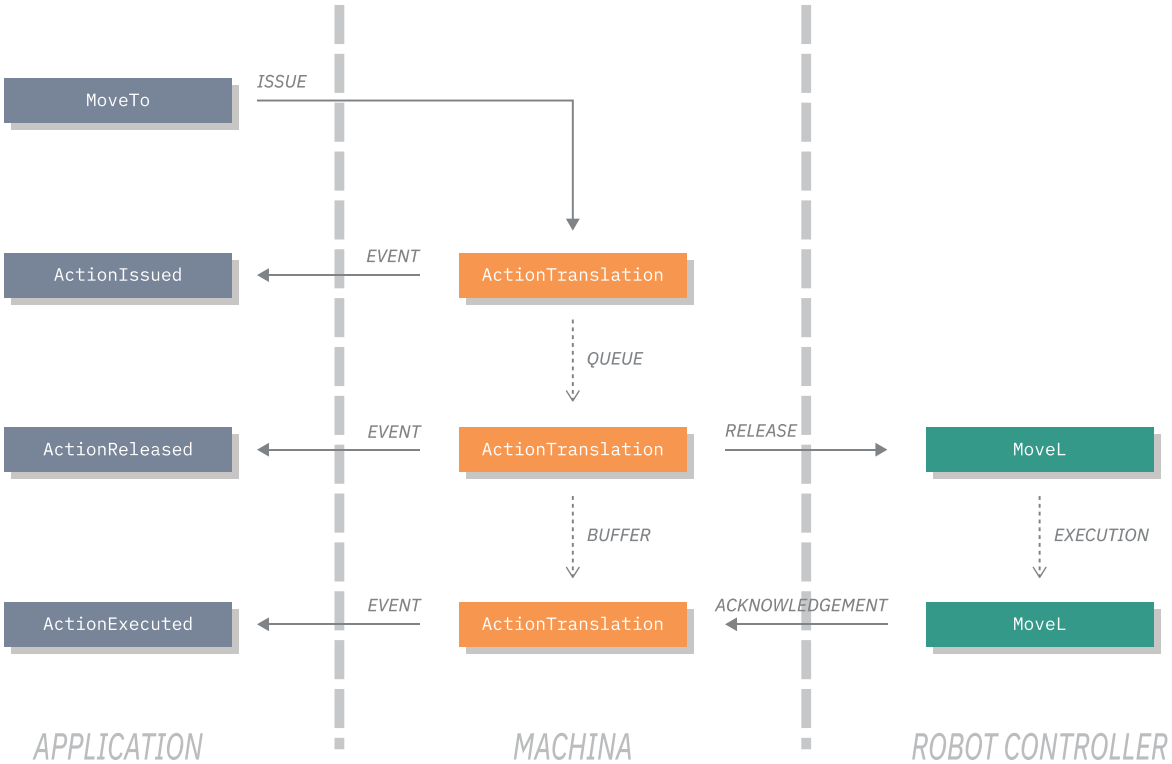

Figure 4: Life cycle of a Machina Action. MoveL represents a move instruction in the device native's language executed by the driver.

simultaneously handle communication, instruction parsing and smooth execution of large programs [39]. Blocks of `blockSize` actions are automatically released to the controller whenever less than `pendingCount` actions are pending to be executed, with `blockSize` and `pendingCount` being customizable. This prevents the device from overflowing with data transfers and memory requirements, and allows for the possibility of on-the-fly modifying or cancelling long programs that have been completely issued but only partially released to the controller. It also facilitates programming styles that are more reactive to the robot execution state. Released actions are immediately applied to the `ReleaseCursor`, which therefore maintains a representation of the state of the robot after execution of all the actions that have been sent to it.

- Once on the device controller, an `Action` is *executed* whenever the changes it represents have been fulfilled by the driver module, and hence the state of the real device reflects those changes. Or, in simpler terms, whenever the robot has completed running that `Action`. On successful execution, the host will receive an acknowledgment message from the driver with information about the action that was just executed. Executed actions are immediately applied to the `ExecutionCursor`, which therefore maintains a representation of the state of the real device right after the last completed action. It is noteworthy that in some instances, and specially with motion actions, program execution on the robot may move on beyond the current instruction even before the robot has fully reached its target position. This is usually the case when the controller tries to smooth the motion trajectory between forthcoming targets. In this cases, the action is considered executed as well.
- Optionally, some devices have the capacity to run multiple threads and send periodic updates on the state of the robot during execution. In this case, such states can be applied to the `MotionCursor`, which therefore maintains the closest representation to real-time tracking of the state of execution, including intermediate states between actions.

Figure 5 is an example diagram of the different cursor representations of the execution timeline in our initial `Hello Robot` program. Assuming that the initial position of the robot is `(400, 200, 500)`, and that Machina is set to stream to the controller in blocks of six actions, this would be the state of the cursors after three seconds of execution:

- The host application executes in a few milliseconds, and is currently halted waiting for user input. Because all actions in the program have been *issued*, the `IssueCursor` is currently at the last action `AxesTo(0,0,0,0,90,0)`.
- The Machina buffer *releases* actions to the robot controller in blocks of six. The state of the `ReleaseCursor` is updated up until `Wait(2000)`.
- As the robot moves at a linear speed of 100 mm/s, three seconds into the program run time the robot has yet to complete *execution* of `Move(0,0,250)`. Therefore, the `ExecutionCursor` is still on the previous action, `Rotate(0,1,0,-90)`.
- If possible, the device updates Machina with the state of motion at small time intervals. Three seconds into the program, the `MotionCursor` would be approximately at position `(400, 300, 600)`.

Feedback

Machina tries to foster programming styles that are reactive to, rather than prescriptive about, the robot execution state, with interactivity, system input and on-the-fly decision making being at the forefront of real-time robotic applications.

The library exposes a collection of `Events` linked to changes in the `Cursors`. These notify the subscribers of the nature of the changes and other useful information. `ActionIssued`, `ActionReleased` and `ActionExecuted` are raised throughout the different

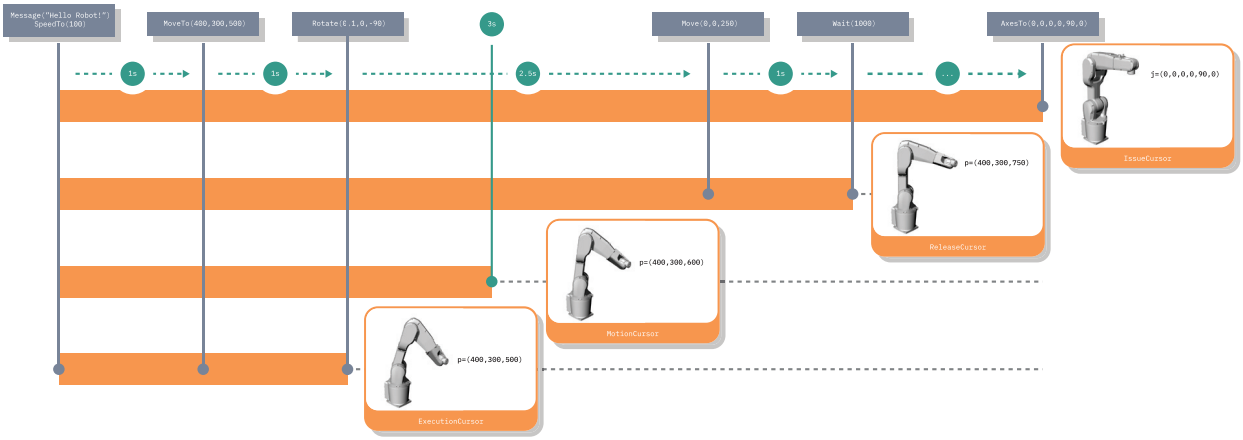

Figure 5: Layered machine states. This diagram shows the different stages of asynchronous execution and Cursor representations for the `Hello Robot` program.

stages of the action execution cycle (see State Model), while `MotionUpdate` is raised anytime real-time information on the state of the robot is received from the controller.

Code Sample 7 shows a small program that starts by issuing motion in a horizontal square 50 × 50 mm loop. Whenever the robot has finished executing an action, `ActionExecuted` is raised. If no actions are pending to be executed, then a new loop is issued, hence creating an infinite loop that can only be interrupted by user input.

Other Features

Machina is designed as an introductory platform to real-time robotics for non-experts; in a way, it is like the 21st century Logo turtle of industrial robots. As such, many of the design decisions made during its development were oriented towards fostering simplicity, intuitiveness, safety and getting applications working right away with minimal setup.

The complete .NET library, as well as other related projects, is fully open source [47]. The aim of this is to open up the field of robotics to curious individuals and power users, either for educational, participation or customization purposes. Furthermore, special consideration is given to thoroughly commenting the source code, to facilitate usage and extensibility of the library.

Additionally, the library has no significant dependencies. It features a full set of custom geometry and robot-related data types, and only references members from the .NET framework. Currently, the only exception is a dependency on the Robot Communication Runtime by ABB Robotics [40], used to interface with ABB controllers and automatically run driver modules in the controller. This dependency is scheduled for deprecation in the nearby future.

Machina.NET is the core library of the `Robot Ex Machina` project [49], an ecosystem of libraries and applications designed around the same core *Enactive Robotics* principles [48], and designed to provide access to real-time robot programming and control from a variety of different platforms. The name is a made-up pseudo Latin expression meaning *robot from the machine*, and it evokes this project's spirit of infusing agency and responsiveness into otherwise passive machines.

Future Work

The Machina project is currently under active development, and significant efforts are being made to improve its functionality across wider range of devices, and make it safer and more robust to use. In particular, current development is focusing mainly on the following aspects:

- KUKA robots: the library currently supports on/offline control of ABB and Universal Robots, and can generate offline code for KUKA robots. However, due to the author's lack of access to the latter for testing, online

Code Sample 7: Whenever the robot has no pending actions left to execute, a new block of actions is issued, generating an infinite motion program.

```
using System;
using Machina;

namespace Sample
{
    class MachinaSample
    {
        static void Main(string[] args)
        {
            Robot arm = Robot.Create("InfiniteLoop", "ABB");
            arm.ActionExecuted += {sender, eArgs} =>

            {
            if (eArgs.Pending == 0) Loop(sender as Robot);
            };

            arm.ControlMode("stream");
            arm.Connect("192.168.125.1", 7000);

            Loop(arm);

            Console.WriteLine("Press any key to finish this program");
            Console.ReadKey();

            arm.Disconnect();
        }

        static void Loop(Robot bot) {
            bot.Move(50, 0, 0);
            bot.Move(0, 50, 0);
            bot.Move(-50, 0, 0);
            bot.Move(0, -50, 0);
        }
    }
}
```

control of KUKA robots is still not available as a feature. Developers willing to extend this functionality to the project are welcome to contact the author or submit pull requests to the project.

- Library of robot models: the library is purposeful generic, with compilers and low-level communication working at the brand level, without the need to incorporate model-specific data. This allows the library to seamlessly connect and control any robot model of a particular vendor. However, this lack of model specifications such as dimensions, joint limitations and mesh geometry makes it impossible to introduce more advanced functionality such as forward/inverse kinematics, out-of-reach computations, singularities, collision detection or visual simulations. While the process of documenting commercial robot models is rather tedious, it could open up important avenues for increased robustness and safety. The future of the library may involve a hybrid model where basic functionality is available for a "generic" robot from a particular vendor, while the above-mentioned advance features might be available if the particular model is available in the robots' library.
- Forward/inverse kinematics solvers: for the reasons explained above, the library currently incorporates no FK/IK solvers, and relies on the robot controller to choose the best configuration for the arm at run time. On top of the limitations explained above, this lack prevents switching between relative and absolute instructions when changing from joint to cartesian space actions, and vice versa. Further work should go into providing suitable FK/IK solvers for the robot models available, in order to overcome these limitations.
- Enhanced safety: as a consequence of the former, the library does not have the capacity to prevent the user from issuing actions that would result in out-of-reach locations, traverse singularities, or collide with known objects. This could potentially pose a safety threat to humans, especially given the novice nature of the users the library is trying to target. The incorporation of robot model libraries and FK/IK would enable safety computations, and the possibility of adding options like `StrictMode` to ensure additional safety measures and robot halting upon known errors.

Safety

It is very important to note that industrial robots can pose a serious safety threat to humans working around them. Robotic actuators are very powerful machines but, for the most part, extremely unaware of their environment; they may not be aware when hitting an object, and continue execution uninterrupted with fatal consequences. Therefore, special attention must be given to the safety of the humans working or interacting with robots, especially if the audience of this project is users who are new to robotics.

In particular, individuals working with real-time robot control should pay special attention to the following measures:

- Be adequately trained to use that particular device.
- Be in good physical and mental condition.
- Operate the robot under the utmost safety measures.
- Follow the facility's and facility staff's safety protocols.
- Make sure the robot has the appropriate guarding in place, including, but not reduced to, e-stops, physical barriers, light curtains, etc.

These guidelines are offered as recommendations, but ultimately, users should follow the protocols in place by their robot/shop managers.

Machina.NET is under active development, and while it will yield warning and error messages on ill-defined actions, due to the current limitations it may not prevent users from executing them. The software is provided "as is," without warranty of any kind, and the author/s should not be liable for any claim or damage arising from its use.

Quality control

Machina has been tested mainly by groups of architecture students and computational design researchers in academic environments. Many of these testers include residents at the Autodesk's BUILD Space in Boston, and participants at teaching-oriented events co-led by the author, such as:

- "Material Systems: Digital Design and Fabrication," [41] a course on robotic fabrication of ceramic systems at the Harvard University Graduate School of Design, Fall 2017 in Cambridge, MA (USA).
- "MindExMachina," [42] a workshop on robotics and machine learning at the SmartGeometry conference, May 2018 in Toronto (Canada).
- "Tight Squeeze: Automated Assembly of Spatial Structures in Constrained Sites," [43] a workshop on robotic construction at the Robots in Architecture conference, September 2018 in Zurich (Switzerland).
- "Talk to a Wall," [44] a workshop on robotic painting with interactive user input and machine learning at the Association for Computer Aided Design in Architecture conference, October 2018 in Mexico City (Mexico).

User observation, feedback, bug reports and feature requests have been crucial in the development of the library and the current level of stability.

The project incorporates a full set of unit tests developed in Visual Studio's `UnitTestFramework`. These tests focus mainly on validity checks for the `Geometry` data types, especially for the conversions between them.

(2) Availability

Operating system

Machina has been tested on Windows 7, 8.1 and 10.

Programming language

C# and the .NET framework v4.6.1.

Compatibility

As of current version, Machina is compatible with the following devices:

Device	Offline	Online
ABB Robots	Yes	Yes
Universal Robots	Yes	Yes
KUKA Robots	Yes	No
Zmorph 3D Printer	Yes	No

Additional system requirements

For online mode, a host computer running a Machina-powered application should be able to establish successful connection with a physical device running a driver module. Simulation tools such as RobotStudio [4] can be used to test Machina with virtual devices.

Dependencies

See "Other Features".

List of contributors

Machina was created and is maintained by Jose Luis García del Castillo y López.

Software location

Archive and Code Repository

Name: Machina.NET
Persistent identifier: https://doi.org/10.5281/zenodo.2579370
URL: https://github.com/RobotExMachina/Machina.NET
Licence: MIT
Publisher: Zenodo
Version published: 0.8.9
Date published: 27 February 2019

Language

English.

(3) Reuse potential

Machina can be used by makers, designers, artists and engineers to create applications that maximize the potential of controlling robots in real-time. Early implementations show the potential of incorporating live feedback in classical one-directional processes, such as 3D printing. The Spatial Print Trajectory project [45, 46] used Machina to stream the print toolpaths of a spatial lattice of 3D printed clay, drive a distance sensor attached to the end effector to measure local deformations on the fresh mixture, and generate the toolpath of the next layer compensating for them. Similar ideas could be implemented for human-robot collaboration in construction, personal fabrication projects, interactive art installations, etc. The author believes that this novel paradigm will open new avenues of research and creative exploration for any individual trying to make things with robots.

Machina is designed for extensibility. The action model ensures that programs can be created in a platform-agnostic way and, by extending the classes that carry device-specific functionality, applied to new robot types beyond the ones available in the library. Documentation on how to do this is maintained in the main repository's wiki, as well as basic use manuals and walkthroughs. Bugs and feature requests can be reported through the repository's issue tracker.

Acknowledgements

I would like to thank Autodesk Inc., the Generative Design Group, the BUILD Space, and in special to Matt Jezyk for his continuous support of this project.

Special gratitude also goes to all the early adopters of Machina whose valuable feedback, input, and very often patience, made this project worthy of sharing with the rest of the world. A special mention to Nono Martínez Alonso for being part of this journey since its very early days.

My acknowledgements to Jonathan Grinham, Saurabh Mhatre, Scott Mitchell and Nate Peters for their time and suggestions at the early stages of this paper.

Competing Interests

The author has no competing interests to declare.

References

1. "Robot, n.1.1." *OED Online*. Oxford University Press, June 2018. Web. 23 August 2018.
2. **Richards, N M** and **Smart, W D** 2013 How should the law think about robots? DOI: https://doi.org/10.2139/ssrn.2263363
3. http://www.abb.com.
4. https://new.abb.com/products/robotics/robotstudio.
5. **ABB Robotics** 2013 *Technical Reference Manual, RAPID Instructions, Function and Data Types.*
6. https://www.kuka.com.
7. https://www.kuka.com/en-us/products/robotics-systems/software/simulation-planning-optimization/kuka_sim.
8. **KUKA Roboter GmbH** 2003 *Software KR C2/KR C3 Expert Programming.*
9. https://www.universal-robots.com/.
10. **Universal Robots** 2018 *The URScript Programming Language, Version 3.5.3.*
11. https://www.universal-robots.com/download.
12. https://robodk.com/.
13. https://www.visualcomponents.com/.
14. **Quigley, M, Conley, K, Gerkey, B, Faust, J, Foote, T, Leibs, J, Wheeler, R** and **Ng, A Y** 2009 "ROS: An Open-source Robot Operating System." In *ICRA Workshop on Open Source Software*, vol. 3 (3.2).
15. **Gershenfeld, N** 2008 *Fab: the coming revolution on your desktop–from personal computers to personal fabrication.* Basic Books.
16. **Chris, A** 2012 Makers: the new industrial revolution. New York: Crown Business.
17. **Mellis, D, Banzi, M, Cuartielles, D** and **Igoe, T** 2007 April. Arduino: An open electronic prototyping platform. In *Proc. Chi* (Vol. 2007).
18. **Baudisch, P** and **Mueller, S** 2017 Personal fabrication. *Foundations and Trends in Human–Computer*

Interaction, 10(3–4): 165–293. DOI: https://doi.org/10.1561/1100000055

19. **Peng, H, Briggs, J, Wang, C Y, Guo, K, Kider, J, Mueller, S, Baudisch, P** and **Guimbretière, F** 2018 April RoMA: Interactive Fabrication with Augmented Reality and a Robotic 3D Printer. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (p. 579). ACM.
20. **Brell-Cokcan, S** and **Braumann, J** (eds.) 2013 *Rob|Arch 2012: Robotic Fabrication in Architecture, Art and Design*. Springer. DOI: https://doi.org/10.1007/978-3-7091-1465-0
21. **Willette, A, Brell-Cokcan, S** and **Braumann, J** 2014 *Robotic Fabrication in Architecture, Art and Design 2014*. Springer.
22. **Reinhardt, D, Saunders, R** and **Burry, J** (eds.) 2016 *Robotic Fabrication in Architecture, Art and Design 2016*. Springer. DOI: https://doi.org/10.1007/978-3-319-26378-6
23. **Willman, J, Block, P, Hutter, M, Byrne, K** and **Schork, T** (eds.) 2018 *Robotic Fabrication in Architecture, Art and Design 2018*. Springer. DOI: https://doi.org/10.1007/978-3-319-92294-2
24. **Andreani, S, Garcia del Castillo y López, J L, Jyoti, A, King, N** and **Bechthold, M** 2012 Flowing matter: Robotic fabrication of complex ceramic systems. In *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction* (Vol. 29, p. 1). Vilnius Gediminas Technical University, Department of Construction Economics & Property.
25. **Seibold, Z, Hinz, K, García del Castillo y López, J L, Martínez Alonso, N** and **Bechthold, M** 2018 Ceramic Morphologies, Precision and Control in Paste-Based Additive Manufacturing. In *Proceedings of the 38th Annual Conference of the Association for Computer-Aided Design in Architecture (ACADIA)*. Universidad Iberoamericana.
26. http://design-io.com/projects/Mimic/.
27. **Gannon, M** 2018 *Human-Centered Interfaces for Autonomous Fabrication Machines* (Ph.D. Dissertation, Carnegie Mellon University).
28. https://www.grasshopper3d.com/.
29. **Schwartz, T** 2013 HAL. In *Rob|Arch 2012* (pp. 92–101). Vienna: Springer. DOI: https://doi.org/10.1007/978-3-7091-1465-0_8
30. **Braumann, J** and **Brell-Cokcan, S** 2011 "Parametric Robot Control, Integrated CAD/CAM for Architectural Design." In *Proceedings of the 31th Annual Conference of the Association for Computer Aided Design in Architecture*. 242–51.
31. https://www.food4rhino.com/app/taco-abb.
32. **Elashry, K** and **Glynn, R** 2014 An Approach To Automated Construction Using Adaptive Programing. In *Robotic Fabrication in Architecture, Art and Design 2014* (pp. 51–66). Cham: Springer. DOI: https://doi.org/10.1007/978-3-319-04663-1_4
33. **Peek, N N M** 2016 *Making Machines that Make: Object-Oriented Hardware Meets Object-Oriented Software* (Ph.D. Dissertation, Massachusetts Institute of Technology).
34. **Bechthold, M** 2010 The return of the future: a second go at robotic construction. *Architectural Design*, 80(4): 116–121. DOI: https://doi.org/10.1002/ad.1115
35. **Landay, J A** 2009 Technical perspective Design tools for the rest of us. *Communications of the ACM*, 52(12): 80–80. DOI: https://doi.org/10.1145/1610252.1610274
36. **Papert, S** 1980 *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.
37. **Reas, C** and **Fry, B** 2007 *Processing: a Programming Handbook for Visual Designers and Artists*. MIT Press.
38. **Steiner, H C** 2009. Firmata: Towards Making Microcontrollers Act Like Extensions of the Computer. In *NIME* (pp. 125–130).
39. **Wijnen, B, Anzalone, G C, Haselhuhn, A S, Sanders, P G** and **Pearce, J M** 2016 Free and open-source control software for 3-D motion and processing. *Journal of Open Research Software*, 4(1). DOI: https://doi.org/10.5334/jors.78
40. https://robotstudio.azureedge.net/install/RobotWare_Tools_And_Utilities_6.07.01.zip. Accessed 23 August 2018.
41. https://www.gsd.harvard.edu/course/material-systems-digital-design-and-fabrication-fall-2017/. Accessed 23 August 2018.
42. https://www.smartgeometry.org/mind-ex-machina/. Accessed 23 August 2018.
43. http://www.robarch2018.org/tight-squeeze-automated-assembly-spatial-structures-constrained-sites/. Accessed 23 August 2018.
44. http://2018.acadia.org/workshops.html. Accessed 23 August 2018.
45. **AlOthman, S, Im, H C, Jung, F** and **Bechthold, M** 2018 Spatial Print Trajectory: Controlling Material Behavior with Print Speed, Feed Rate, and Complex Print Path. In *Robotic Fabrication in Architecture, Art and Design (Rob|Arch).* Springer. DOI: https://doi.org/10.1007/978-3-319-92294-2_13
46. **Im, H C, AlOthman, S** and **García del Castillo y López, J L** 2018 Responsive Spatial Print: Clay 3D Printing of Spatial Lattices Using Real-Time Model Recalibration. In *Proceedings of the 38th Annual Conference of the Association for Computer-Aided Design in Architecture (ACADIA)*. Universidad Iberoamericana.
47. https://github.com/RobotExMachina. Accessed 23 August 2018.
48. **Garcia del Castillo y López, J L** 2019 *Enactive Robotics: An Action-State Model for Concurrent Machine Control* (D.Des. Dissertation, Harvard University). http://nrs.harvard.edu/urn-3:HUL.InstRepos:41021631.
49. **García del Castillo y López, J L** 2019 Robot Ex Machina: A Framework for Real-Time Robot Programming and Control. In *Ubiquity and Autonomy, Proceedings of the 39th Annual Conference of the Association for Computer-Aided Design in Architecture (ACADIA)*. University of Texas.

How to cite this article: García del Castillo y López, J L 2019 Machina.NET: A Library for Programming and Real-Time Control of Industrial Robots. *Journal of Open Research Software,* 7: 27. DOI: https://doi.org/10.5334/jors.247

Submitted: 19 September 2018 **Accepted:** 05 July 2019 **Published:** 13 August 2019

]u[*Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press

OPEN ACCESS