Mech - A Programming Platform for Robots and Other Data-Driven Systems

Corey Montella
cmontella@cse.lehigh.edu
Lehigh University
Bethlehem, PA, United States

Abstract

We present Mech, a programming language and system for building data-driven systems such as robots, games, simulations, animations, and user interfaces. Mech aims to increase developer productivity and lower the barrier of entry to programming by replicating the expressiveness of Matlab for robot programming while offering the platform features of the Robot Operating System (ROS), and the performance of C++. Mech leverages unique and expressive abstractions suited for robot control tasks, allowing for concise specification of complex systems with potentially a two-order-of-magnitude discount in code length compared to typical code. Mech takes into account the highly parallel, asynchronous, and distributed nature of robots and other data-driven systems, which is not well-served by existing programming languages. The paper maps the design of Mech in the programming system design space, presents two examples of its utility for simulation and robot programming, and shows it its performance is on par with compiled languages like Rust.

CCS Concepts: • **Computer systems organization → Robotics**; • **Software and its engineering → Data flow languages**.

Keywords: programming language,robotics

ACM Reference Format:
Corey Montella. 2023. Mech - A Programming Platform for Robots and Other Data-Driven Systems. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Onward!).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

Onward!, October 22–27, 2023, Cascais, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The robot revolution roboticists have been presaging is finally coming to pass; robots are vacuuming floors, mowing lawns, checking stock in stores, managing warehouses, delivering goods to consumers, and more at unprecedented levels [18]. We are at the precipice of a new era of robotics, where the impact they have on ordinary people will perhaps be just as impactful as that of the Internet in the 1990s.

As we stand at that precipice, the complexity of robots is exploding such that the general population, who will soon be working with and walking among robots every day, are being left further behind. Because of this, many are concerned that a greater prevalence of robots and automation in society will create societal disruption as they replace workers. If robots are meant to be such an integral part of our lives, then everyone should have the access, opportunity, and knowledge to program them. But with today's state-of the art robot systems coming in at perhaps a billion lines of code [13], is there any hope that the development of such complicated machines ever be democratized?

The 2007 DARPA Urban Challenge (DUC) offers an informative example that may bring some hope. The DUC was the third in a series of competitions hosted by DARPA aimed at facilitating American development of autonomous vehicles for both domestic and military purposes. The competition tasked researchers with creating autonomous vehicles that could navigate an environment complete with sidewalks, intersections, lane markings, and other features common in modern urban traffic scenarios. DARPA provided two competitive tracks during the DUC: they selected six Track-A teams to receive $1 million grants to fund their entry to the competition; meanwhile five Track-B teams were allowed to compete, but were not provided funding from DARPA. The results of the race show that six teams out of the 11 total managed to finish the course, and out of those six only one was a Track-B team – the Ben Franklin Racing Team* (BFRT) [4]. Every other finishing team was Track-A. How did the BFRT manage to do this with a quarter of the funding used by Track-A teams?

The answer may lie in how many lines of code the BFRT wrote for the challenge; whereas MIT, Stanford, CMU, and other top contestants clocked their codebases in at hundreds-of-thousands of lines of C++ code [20], the BFRT wrote a mere 5,000 lines of Matlab code [3]. Given these two systems

[publisher.py]

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1

def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

[subscriber.py]

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)

def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

[publisher.mec]

```
#count = 0

Publish the message
  ~ 2<Hz>
  #msg = "Hello World! " + #count
  #count :+= 1
```

[subscriber.mec]

```
"I heard: " + #msg
```

Figure 1. (Top) A simple publisher subscriber is implemented in ROS and Python, where one program (a publisher) sends data to another program (a subscriber) over a ROS network. Highlighted in boxes are the irreducibly complex parts of the program. All other code is incidental. (Bottom) The same publisher subscriber implemented in pure Mech. Notice that the Mech version is almost entirely essential aside from a comment.

of similar behavior, the excess code required to specify the system in C++ represents what Mosely and Marks would term "incidental complexity" [22], which is any complexity involved in specifying the system that is not directly related to the "essential complexity" of the problem domain. I.e., every problem is essentially complex, and there's no way to elide that complexity without respecifying the problem. But not every system implemented to solve a particular problem is equally complex – some are more complex than others, and while that extra complexity may be justified, it comes at a cost. For lines of code on a robot, this complexity manifests as time spent writing, maintaining, and debugging all that code. Figure 1 shows the kind of dramatic code-length reduction we can expect when incidental complexity is minimized.

Thus, we can frame our objective: to increase roboticist productivity and lower the barrier of entry to robot programming, we can do what the BFRT did in building their robot, Little Ben – leverage abstractions especially suited for robot control tasks. This will allow us to concisely specify complex robot systems, with potentially a two-order-of-magnitude discount in code length. In so doing, we will decrease the incidental complexity involved in robot programming, allowing us to create ever-more complex research and commercial robots, while simultaneously lowering the barrier of entry for students and hobbyists building simpler machines.

Equipped with this bit of history and a clear goal, we present Mech, a programming language and platform that aims to replicate the expressiveness of Matlab for robot programming, while also offering the features that roboticists enjoy from platforms such as the Robot Operating System (ROS). Its intended applications beyond robotics are games, simulations, animations, and user interfaces. Our goal with Mech is to use unique and expressive abstractions to increase developer productivity, and lower the learning curve of robot programming. Our hope is that with Mech, what's possible for a graduate student with current tools would be doable for an undergraduate; and what's doable for an undergraduate today with current tools, would be doable for a high or middle school student with Mech. To paraphrase Alan Kay, with Mech, simple things should be simple while complicated things should be doable.

We believe such a dramatic speedup in developer productivity is possible due to the presence of a severe impedance

mismatch between current robot programming tools and the nature of robotics engineering/research; the key insight driving the design of Mech is that robots have a highly parallel, asynchronous, and distributed nature – yet many of the tools we use to create them, like C++ and Python, were designed in an age when computers had at most a single core and were marginally networked. The design of these languages reflects the architecture of the machines of their time, and consequences of these designs carries through to today, despite the underlying machines having changed substantially. For the first time in the history of computing the exponential progress of CPU core frequency has stalled, but an explosion of core counts in consumer-grade processors has begun instead. At the same time, "co-processors" are proliferating to handle specialized workloads like graphics, artificial intelligence, computer vision, and physics. These processors are especially well-suited for robotics applications, as these are all core components of modern robot systems.

In this paper, we detail the design of the Mech programming language, and present several examples that illustrate its utility. First, we briefly survey existing robot programming languages in Section II. Then in Section III we provide an overview of Mech's syntax and semantics. Section IV catalogs the technical dimensions of the Mech programming system. Finally, we evaluate Mech's performance against several programming languages for two representative tasks: running a simulation, and a robotics algorithm. Finally, in Section V we conclude with a roadmap for Mech and a call to join our cause of democratizing robotics for everyone.

The Mech project was started in 2014 in response to a perceived limitations of existing programming languages used in the field of robotics research. Since then, the project has undergone several significant milestones and achievements.

In 2018, the development we began implementing the language, drawing inspiration from various programming languages, including Eve, Smalltalk, Logo, Lucid, Matlab, Excel, and ROS. The project's development was a collaborative effort among experts in robotics research and programming language design.

As of now, Mech is in the beta-stage of development, and has over 6,000 commits and a small core budget of 10k LOC (currently at 12k). The team implemented Mech in Rust, a fast and safe systems language, and designed it to be safe, efficient, and accessible to a broad range of users in the field of robotics research. Mech is open source and licensed under Apache 2.0.

Though Mech has been primarily presented within the context of robotics so far, this paper will demonstrate that its application extends beyond that; Mech's programming paradigm, designed for data-driven systems, can be effectively applied to various other domains, such as games, animations, and user interfaces.

1.1 Language Purpose

The purpose of the Mech project is to reduce development time of robotic systems by leveraging abstractions that harmonize with robot architecture and design patterns, and which are unavailable or cumbersome to access in languages that are currently in widespread use.

To achieve these goals, Mech's design philosophy centers around data flow programming and asynchronous programming. This approach enables efficient and scalable solutions to complex robotics problems while providing high-level abstractions that make it easier for beginners to get started with robotics programming.

1.2 Design Philosophy

Mech's design philosophy is tailored to the unique needs of robotics development, which are distinct from those of traditional systems that process static data. Robots have agency, can move and interact with their environment, and operate in real-time. Therefore, it's logical to start from the nature of a robot and select the features that best align with its requirements. The combination of features in Mech creates a singular point in the programming system design space that addresses these needs effectively.

Although none of Mech's individual features are novel in themselves, the combination of these features in a single language is what makes it unique. The purpose of this paper is to explore the comparatively unexplored territory of robotics programming languages, and describe how Mech's features interact across the various axes of the design space. By doing so, this paper aims to establish Mech as a benchmark for programming languages designed for robotics development.

1.3 Major System Features

In this section, we list the key features that summarize Mech's character and set it apart from other languages. For a detailed discussion on these features, see Section 4.

- **Mech is a dataflow language** – computation happens in the presence of data, and the system is idle without it. Mech programs are expected to resemble a control loop that runs forever, although traditional programs that terminate after processing a batch input can be written.
- **Mech is reactive** – all computations are kept up to date as dependent data change. This is especially relevant for robots whose architectures are designed as control loops that continuously consume sensor data and react to it.
- **Mech is parallel** – The programming metaphor in Mech is that everything is a table, much like in Matlab. Functions and arithmetic, logic, and comparison operators in Mech are broadcast over the elements of the table, so most computations can be parallelizable by default.

Define Table	`1 or [1]`	1x1 scalar
	`[1 2 3]`	1x3 row vector
	`[1;2;3]`	3x1 column vector
	`[1 2` ` 3 4]`	2x2 matrix
	`[a: 1, b: 2]`	Inline table
	`[|a<u64>|]`	Empty table

	`[|a b|`
	`  1 "x"`
	`  3 "y"]`
		Heterogenous rows

	`[1; "x"]`	Heterogenous columns
	`[a: [b: 2]]`	Nested tables
	`x = 1`	Define local variable
	`#x = [1 2 3]`	Define a global table
	`#x = #y`	Define reactive relationship
Select Element	`x{1}`	First element of variable
	`x{1,2}`	Row 1, column 2
	`x.a{1}`	Column a, row 1
	`x{:,1}`	Every row in the first column
	`x{1,:}`	Every column in the first row
	`x{ix}`	Filter elements according to ix
	`x{1}{2}`	Nested tables
	`x.a.b`	Dot select nested tables
	`x.a,b,b`	Swizzle columns a, b, and b

Change Table	`x{1} := y`	Set the first element of x to y
	`x{1} :+= y`	Add y to the first element of x
	`x += [1 2 3]`	Append a row to x
	`x += [a: 1]`	Append a partial row to x
	`x >- y`	Split table y into x
	`x -< y`	Flatten nested table y into x
Operators	`x * y`	Multiply x and y element-wise
	`x >= y`	Compare x and y element-wise
	`x | y`	Logical Or x and y element-wise
	`x ** y'`	Matrix multiply x and y transpose
Temporal Operators	`~ x`	Whenever x changes, execute block
	`|~ x`	As soon as x, execute block once
	`~| x`	Until x, execute block
	`#x <~ y`	Async assign y to #x
	`#x = 1 ~> #x + 1`	Generate a stream
Units	`x = 10<u64>`	Define an unsigned 64-bit integer
	`x = 1<m> +` `  2<m/s> * 3<s>`	Do mathematical operations on quantities. This evaluates to 7m.
Functions	`[x] = add(y<f32>)` `  x = y + 2`	Define a function
	`x = add(y: 20)`	Evaluate a function

Figure 2. Mech syntax overview. This currently represents the entirety of the language. There are no control flow structures in Mech. Note that the Mech programming system is distinct from this syntax, and that any number of syntaxes or graphical programming interfaces could be attached to a Mech runtime. Currently there are no other syntaxes available, but this is as potential area of expansion.

- **Mech is distributed** – programs in Mech define a network of nodes communicating via message passing. This is the dominant paradigm in mid-level general robotics programming, and this concept is built into the ambient semantics of the Mech language. The Mech runtime automatically figures out the topology of a program's compute network based on data dependencies, or a programmer can specify one manually within the language itself.
- **Mech is asynchronous** – in interacting with the physical world, robots are inherently asynchronous machines. Therefore, Mech embeds this concept into the language semantics. Because the entire language is distributed, all library calls are asynchronous. This means opening a file or sending a network request is nonblocking by default; a Mech program that makes a request to a network server will continue running and working on other tasks while it waits for the response, and then will react to it as soon as it arrives.

1.4 Target Users and Applications

Mech should be usable by industry professionals, academic researchers, and students as young as middle school. Although these users have a wide difference in programming ability, We believe the impact of the Meh system design will serve to significantly lower the barrier of entry to robot programming for students. For experts, we hope they can use Mech to augment their existing skills and knowledge.

- **Industry Professionals:** Experienced engineers need a safe and efficient language that integrates with their existing systems to design and develop robots.
- **Academic Researchers:** Experienced researchers require an open-source language that enables easy collaboration and helps create robotic systems for advanced research.
- **College Students:** Students need a user-friendly language that is easy to learn and use, while allowing them to work with robotic systems.
- **Middle School Students:** Beginners need an engaging, user-friendly language that allows them to develop programming skills while having fun and creating interactive robotics projects for STEM education.

We also intend Mech to be used beyond just robotics; any system which can be described or modeled as a data-driven feedback control loop would be well-suited for Mech. This system design covers a surprising variety of problem domains:

- **Games:** reactive programming model, built-in graphics and audio libraries, and native GPU support make it well-suited for game development.
- **Scientific Computing:** support for physical units and automatic differentiation make it well-suited for scientific computing applications, such as physics simulations and machine learning.

- **Control Systems:** dataflow nature and feedback control loop model of Mech programs make it well-suited for control systems generally, such as those used in industrial automation and process control.
- **Real-Time Systems:** temporal operators and predictable, allocation-free execution make it well-suited for real-time systems, such as those used in avionics and automotive applications.
- **Web Applications:** ability to compile to WebAssembly makes it well-suited for web applications, particularly those that require high performance computation.

1.5 Syntax Primer - The Bouncing Balls Example

Figure 2 provides an overview of the Mech syntax, and Fig. 3 demonstrates an implemted simulation of 2D balls in a bounded arena while applying gravity and repulsion. The Mech code is embedded in a Markdown dialect called "Mechdown" that enables literate programming in plain text.

The Mech code starts by defining tables that include the initial position and velocity of the balls, the acceleration due to gravity, the time step, and the boundaries of the arena. These tables are annotated with physical units that ensure consistency in calculations and scale compatible units when necessary.

A timer is set up on line 11 to update the positions of the balls every 16 milliseconds. Asynchronous code starting on line 13 waits for the timer to change and recomputes the positions of the balls accordingly, using their velocities and the time step.

Finally, the code on line 20 enforces boundary constraints on the "balls" table to prevent any ball from leaving the arena. If a ball exceeds the boundary, its position is set to the boundary before the computation resolves. Logical indexing is used to apply the constraint only to the affected rows. This ensures that the balls never exist in an invalid state.

```
1  Bouncing Balls Simulation
2  =========================
3
4  This program is a simulation of three balls in a
.  bounded arena. Each of the balls are accelerated
.  by gravity and are repelled by the bounds of the
.  arena.
5
6  In Mech, code and prose are interwoven. This
.  listing is a valid Mech program (indeed, the
.  source of this entire paper is itself a valid
.  Mech program). Prose is expressed in a dialect of
.  Markdown called Mechdown.
7
8  #balls = [|x<m> y<m> vy<m/s> vx<m/s>|
9             20   10   0       0
10            100  50   0       3
11            300  100  0      -5]
12 #gravity = 9.8<m/s^2>
13 #dt = 16<ms>
14 #bounds = [x<m>: 500 y<m>: 600]
15
16 A table preceded with a hashtag belongs to a
.  global network-wide scope, accessible from any
.  node in the Mech compute network. Writing to and
.  reading from these tables can be thought of as
.  message passing.
17
18 #time/timer += [period: 16<ms>]
19
20 The above statement defines a timer by adding a
.  row to the table #time/timer. This table is not
.  defined in our program, so Mech will search for
.  it in the Mech package repository, and
.  automatically download it if it's available. If
.  not, Mech will search for it on its internal
.  network and wait until it becomes available on
.  another node.
21
22   ~ #time/timer.ticks
23   #balls.x,y :+= #balls.vx,vy * #dt
24   #balls.vy :+= #gravity * #dt
25
26 Code that is indented two spaces from the margin
.  runs as an asynchronous block. The preceding
.  block waits for the timer to change and then will
.  recompute the positions of the balls when it
.  does. This feature is what makes Mech a reactive
.  language like Excel, which recomputes all values
.  as their dependencies change.
27
28   ~ #time/timer.ticks
29   ix = #balls.x,y > #bounds
30   ixx = #balls.x,y < 0
31   #balls.x,y{ix} := #bounds
32   #balls.x,y{ixx} := 0
33   #balls.vx,vy{ix | ixx} := -#balls.vx,vy * 80%
34
35 This block of code enforces boundary constraints
.  on the "balls" table to ensure that no ball can
.  ever leave the bounds of the arena. If any do,
.  their position is set to the bounds before the
.  computation resolves, so that the balls never
.  exist in an invalid state. Logical indexing is
.  used to apply the constraint only to rows which
.  violate it.
```

Figure 3. Mech program modeling a physical system.

2 A Brief Survey of Robotics Programming Languages

Robot programming languages have a long history that closely tracks the history of robotics in general – as robot designs have become more varied, and their capabilities have expanded over the years, robot programming languages have evolved to meet the control challenges they create. We can roughly divide the robot programming language field into three broad camps: industrial languages, general purpose languages, and educational languages. We also survey languages with similar features to Mech used in programming robots.

2.1 Industrial robot languages

In the early history of industrial robots, languages were often purpose-built by robot manufacturers to be used with their line of robots. This tight coupling of robot and language meant that companies could offer specialized interfaces to

their hardware which could take advantage of their product's unique offerings. However, from an industry perspective it meant that skills in programming one robot often could not be transferred to another robot. Many of these languages – such as Kuka's KRL, FANU's Karel, Yasakawa INFORM II, or ABB's RAPID language – were descended from Pascal and therefore were largely procedural. Programs often involved a sequence of instructions to set the robot's various joints to a specified angle [2][9].

Beyond textual programming languages, industrial robots are often programmed using a device known as "teach pendant", which provides a convenient and simplified user interface to a robot without the need to write code. This allows a technician to physically program a robot by directing it with the teach pendent, and then recording the instructions for replay later.

2.2 General purpose robot languages

Robot requirements differ greatly outside of settings where environmental conditions cannot be predetermined, Robots that navigate in these environments rely on an ever-expanding variety of sensors and actuators to measure and respond to their surroundings. Accordingly, robot languages are significantly more complex in this arena. Languages here can be divided into three camps: low-level hardware control, mid-level behaviors, and high-level planning.

For programming low-level control, languages exist that interface directly with hardware. It is quite common for the C programming language to be used in these contexts due to its ubiquity. Concerns in this domain involve interfacing with hardware directly, so the ability to manipulate data and memory at the bit level is important here.

At the mid and high levels, the current state of robotics software development revolves around the Robot Operating System (ROS) platform. ROS is a middleware that provides various tools to facilitate robotics programming, such as interprocess communication, logging, and data visualization. With its robust message passing capabilities, ROS encourages a node-based architecture where loosely coupled functional units within the robot system (nodes) pass messages to one another.

Not being a programming language itself, the ROS platform leverages general purpose programming languages to implement the logic contained within each node. Languages such as C++ and Python are among the most popular in robotics today, but each suffer critical shortcomings in implementing asynchronous distributed systems (hence the need for the ROS middleware in the first place). Several other languages have been used in this domain as well [15][5].

2.3 Educational robot languages

At the other end of the complexity scale are educational robot programming languages. These languages are designed to appeal to children and novices, and are often presented with colorful and animated, cartoonish interfaces. Scratch [21] and Blockly are two block-based languages that attempt to hide the complexity of programming by wrapping the various elements in Lego-like blocks that programmers snap together using a colorful GUI. However, this only hides the complexity and has limited utility outside of learning contexts.

Seymour Papert took a different approach when he developed the Logo language, which preceded Scratch and Blockly. He designed Logo to account for child psychology to make programming easier. In Logo, programs are written in the context of a graphical drawing interface, and programmers assume the perspective of a "Turtle" that serves as the in-program avatar of the programmer. Commands are issued to the Turtle like "forward 100" to move the Turtle forward 100 pixels. The Turtle was the result of incorporating research from childhood development into the design on the language. The key insight Papert leveraged is that while young children may not have a well-developed ability for abstract reasoning yet, they do have excellent reasoning over their own motor skills. By asking them to imagine themselves as the Turtle, Papert found he could get young learners to perform deep abstract reasoning tasks through the lens of their own kinematics, with findings showing that they were able to come to conclusions about physics related problems usually reserved for graduate studies. The lesson of Logo is that abstractions targeted at a particular domain or even a particular user psychology can allow novice programmers to nevertheless write sophisticated programs [25].

Other efforts exist in the educational robot programming sector, including Mindstorms from Lego, which combines Lego building sets with a computer and sensors. The programs are written in a visual, flow-based language, or a dialect of C, which is common in toy-grade robotics products.

2.4 Distributed Robot languages

The literature presents a diverse array of programming languages and paradigms designed to manage complex, distributed systems such as robot ensembles and reactive networks. Meld [1], for instance, utilizes a logic-based approach to program a collection of robots from a global perspective, contrasting with Mech's use of an iterative approach based on sensor data and internal models. However, Meld does not offer Mech's time-travel debugging, error handling, or access control mechanisms.

DREAM [17], on the other hand, emphasizes the balance between consistency and overhead in distributed reactive programming. While sharing with Mech the concept of dynamic reactivity, DREAM's focus is on adjusting consistency levels for different applications, a feature not explicitly present in Mech.

Notably, a series of papers focus on fault tolerance and reactive programming. One work presents an approach to

provide fault tolerance for distributed reactive programming without changing existing programs' behavior, aligning with Mech's ability to handle errors [19] . Meanwhile, another discusses eliminating "glitches" or temporary violations of data flow invariants [27], which is also a something that Mech's support by way of a built-in transactional database.

Myter et al. identify an essential disconnection between reactive programming languages and the actual requirements of reactive distributed systems [23]. They argue that existing approaches are not adequately equipped to handle reactive distributed systems because they are based on centralized coordination points and overlook partial system failures. In contrast, Mech is designed to operate without a single central point of coordination and acknowledges the possibility of partial failures, making it well-suited for complex and distributed systems.

Buzz, like Mech, is designed for heterogeneous robot systems, but it focuses on swarm dynamics [26]. ASEBA [16], similarly, proposes a modular architecture for event-based control of complex robots, while SYNDICATE [8] presents a coordinated, concurrent programming language. While these approaches offer intriguing solutions for specific scenarios, none seem to encapsulate Mech's combination of sensor data processing, error management, flexible access control, and other features.

3 Language Design

Mech's programming model relies on a feedback control loop, where it waits for data to update a compute network. Unlike traditional languages with entry points like a "main" function that starts execution and runs until completion, Mech programs run indefinitely, waiting for data and reacting to it. There is no need to invoke a starting function; instead, you supply the runtime the relevant data needed to start the desired process.

The key ingredients to a Mech program are values, tables, blocks, cores, and machines. Values are the most concrete element in Mech, whereas machines are the most abstract.

Mech programs are built on a unified value type (number, string, Boolean) as basic data elements. Values are organized into rows and columns and represented as tables for structured data storage. Tables are the ubiquitous and only data structure in Mech, that can represent scalars, vectors, matrices, and tensors as well.

Blocks, containing self-contained transformations, interact with tables to process data. A cores encompasses any number of blocks, acting as a computational engine that manages memory and block order execution.

Finally, machines package and distribute cores, providing modular and reusable functionality. This hierarchical organization, from values to machines, results in a modular, maintainable, and scalable programming language for various applications.

3.1 Programming Model

Mech is ideal for systems using asynchronous input streams from various sources. relying on sensory data from cameras or accelerometers. Robots utilize numerous sensors like cameras, IMUs, and LiDARs as inputs. As these sensors produce data, Mech processes it to update a model. This model then issues controls to actuators, creating a continuous cycle of sensory input, processing, and response.

Similarly, in real-time games like pong, Mech updates the ball's position every tick of the game clock. Player actions drive game events like moving the player paddles.

For desktop applications with graphical interfaces, Mech processes user inputs like mouse movements and button presses, updating the display with computed program elements on each command, otherwise maintaining a static ("immediate mode") interface.

In these scenarios, Mech only computes when data is present. After data arrives, Mech triggers computations and updates the state, then the system waits for more data. This constitutes a Mech program's lifecycle.

In the following sections, we will build the Mech language up from it's base elements, describing the fundamental atoms (values) to the highest abstraction (machines).

3.2 Values

Mech programming language supports a variety of data types that facilitate the creation of complex data structures and their manipulation through a set of operators and indexing primitives. This section discusses the basic data types available in Mech, including numbers, strings, booleans, and tables, as well as their representations and use cases.

3.2.1 Numbers and Quantities. Mech provides support for numerous number literal formats, such as decimal (10, -42), hexadecimal (0x01234ABCDEF), octal (0o77), binary integers (0b1010), floating-point numbers (3.14, -1.23), scientific notation (6.02e23), and complex numbers (2.5+2i). Decimal numbers without a type specifier default to f32, but other kinds can be indicated using an annotation (e.g., 123<u8>). In addition to basic number literals, Mech accommodates numbers with physical units, known as quantities (e.g., 5<m> for 5.0 meters). Mech can automatically handle unit conversions during arithmetic operations, such as adding 5<m> and 10<ft> to obtain 8.0484<m>. Mech supports various numeric kinds, including unsigned and signed integers, and floating point (IEEE 754-2008).

3.2.2 Strings. Mech employs UTF-8 encoding to enable support for Unicode and emojis within its string data type. This is crucial in facilitating the use of various languages by default. Additionally, Mech's string type is interned as 64-bit IDs into the system, and once defined, they are immutable.

3.2.3 Booleans. Boolean values in Mech are represented by the keywords "true" and "false". These represent the only

English keywords in the syntax, so to support a generic Mech, we also support the Unicode characters such as check mark and x mark, to represent true and false. Boolean values can also be the result of boolean expressions or logical operations. They are particularly useful for filtering tables with logical indexing.

3.2.4 Table Literals. Mech's table data type consists of rows and columns, where each column represents a specific attribute of the data, and each row represents a specific instance of the data. Tables can be created using literals, which are indicated by square brackets [] enclosing a list of values, with spaces and/or commas delineating columns and semicolons and/or newlines delineating rows. For example, [1 2; 3 4] represents a 2x2 matrix of numbers. Index aliases can be added to table columns to improve readability, as shown in [|name age| "Yan" 20; "Seth" 23]. Kinds can be assigned to table columns using a kind annotation, such as [|name<string> age<u8>| "Yan" 20; "Seth" 23]. Mech also supports inline tables, like [name: "Yan" age: 20], and nested tables, which allow for the creation of more complex data structures such as trees and structs, for instance, [type: "div", contains: "Hello World!", parameters: [width: 100 height: 50]] could represent an HTML div element.

3.3 Tables

Mech represents all variables as tables, which are highly flexible and can be used to represent everything from scalars to tables to arrays to vectors. Tables are assigned to identifiers using the = operator, and they have block scope by default but can be made globally scoped by prepending the identifier with a # symbol.

3.4 Table Scoping

Mech allows the definition of both local and global variables in a block. Local variables can only be accessed within their own block, while global variables are accessible throughout the program. For instance, #pi = 3.14 is a globally scoped variable that holds the value of pi.

3.5 Kind Annotations

Mech tables can be annotated with a kind, which specifies the expected type of the table contents. For example, #num<u64> = 1234, #str<string> = "Hello, world!", and #bool<bool> = true are annotated variables. The string and bool annotations can be inferred from the assigned datatype, but the u64 annotation is necessary because the inferred datatype of the number literal 1234 is f32. Kind annotations provide a more precise typing system and help avoid common programming errors.

3.6 Access Control

Mech's architecture supports high-level and low-level access to data and operations, with access control enforced dynamically by cores. High-level access involves abstract operations such as process control (starting / stopping cores) or parameter adjustments, requiring specific permissions for safe execution. Low-level access covers fundamental operations, including direct hardware interactions (e.g. reading files, writing to the network) and resource management (e.g. allocating memory, accessing processors like GPUs).

Mech cores enforce capabilities by checking if the token associated with requested operation has the necessary capabilities and a signature which is validated against a public key. This real-time authorization mechanism ensures that every action performed is authorized.

3.7 Blocks

Blocks are the fundamental building blocks of Mech programs, allowing for the creation, selection, transformation, and writing of data. They are composable, orderless, and reactive units of code, which makes them highly flexible and adaptable for various programming tasks. Blocks are indicated by indenting code from the margin. Contiguous lines are compiled together as a block of transformations on the tables named in the block.

3.7.1 Blocks are Composable. Composability in blocks refers to their ability to depend on other blocks and to be depended upon by other blocks. This creates a data flow between blocks, which is determined by their data dependencies and productions. As data gets updated, blocks automatically re-execute to update their results, making it easy to build modular programs and maintain complex data pipelines.

3.7.2 Blocks are Orderless. The order in which blocks are written does not affect the computation; Mech determines the correct ordering of blocks based on their data dependencies, which allows for greater freedom in exploratory and expository programming without being constrained by an arbitrary block order.

3.7.3 Blocks are Reactive. Reactivity in blocks ensures that their results are automatically updated as the data they depend on changes. This feature applies to the entire block graph, making it easy to build dynamic and responsive applications without worrying about the details of how the computations will be updated as data changes.

3.8 Cores

Mech cores serve as the backbone of the Mech programming language, providing encapsulated computational engines that efficiently host blocks, tables, and manage memory allocation. They support asynchronous functions and machines, enabling concurrent execution of tasks, and facilitate distributed programming by connecting to other cores through core networks.

Mech cores are self-contained computational units designed to host blocks and tables, with each core allocating its own memory arena for storing the database. The encapsulation ensures that each core operates independently and efficiently while allowing for concurrent execution of tasks. This takes full advantage of available hardware resources and leads to improved performance.

3.8.1 Distributed Programming with Cores. One of the key features of Mech cores is their ability to connect and communicate with other cores. This is achieved through a core network implemented on top of UDP sockets and websockets for integration with browsers. This networking capability enables users to create distributed programs where cores can send and receive data from other cores, allowing for complex, collaborative data processing across multiple cores.

The ability to connect with other cores and engage in distributed programming enables users to leverage the power of multiple cores working together. This collaboration results in efficient resource utilization and improved performance for large-scale applications.

To ensure the security and integrity of distributed programs and their data, Mech offers a capability permission system that manages and protects network resources. This system allows users to control access to resources, ensuring that only authorized cores can access and manipulate data. By maintaining strict access control, Mech helps to preserve the security of the distributed program and its data, while still enabling seamless collaboration and communication between cores.

3.8.2 Security and Resource Management.

3.9 Machines: A Comprehensive Module System

Machines, a cornerstone of Mech's architecture, serve as a robust module system. Their role extends beyond packaging and distributing cores for reuse. They promote maintainability, scalability, and interoperability in Mech's program design. Unlike traditional libraries, machines encapsulate more than functions or data structures; they have the capacity to encapsulate entire applications. This distinctive functionality allows the sharing of complex features, data structures, or even complete applications across programs.

Machines come equipped with several key features such as automatic dependency resolution and the ability to download from Mech's machine repository. They support not only native Mech functions but also functions compiled from other languages.

3.9.1 Interoperability and Extensibility. Mech supports interoperability by accommodating Machines that contain functions compiled from high-performance languages such as Rust. Mech also supports JavaScript through WebAssembly (Wasm), ROS, and C++. This cross-language support enables Mech programs to leverage the performance, safety, and ubiquity of these languages, thereby enriching the Mech environment with diverse capabilities.

This broad integration also gives rise to an extensible module system that allows users to augment Mech's native functionality with specialized libraries and tools from various ecosystems. This capacity allows users to draw on the strengths of external programming environments while capitalizing on Mech's unique features.

3.9.2 Capabilities. Machines in Mech, whether started independently or loaded into an existing core, operate in tandem with the system's capabilities, ensuring system integrity and access control. Each machine has a set of necessary capabilities to function, which the corresponding core must possess for the machine to load successfully.

An example of this process involves the io/write machine. Before this machine, which manages the program's ability to write to the standard output (stdout), is loaded, the core checks if it has the appropriate, valid, and signed capability to write to stdout.

4 Technical Dimensions

Understanding a programming language's syntax and semantics is usually enough to comprehend its purpose and capabilities in relation to other languages. However, Mech is more than just a language, as it includes built-in features for program distribution and debugging. Thus, Mech requires a more comprehensive taxonomy to describe the entire system.

In this section, we will describe Mech as a programming system by using the technical dimensions proposed in [14], which provide a framework for analyzing programming systems. Programming systems encompass a broader range of tools and approaches beyond just the language, and by examining Mech within each technical dimension, we can compare its design choices to other programming systems and determine where it falls within the spectrum of possible design choices.

4.1 Interaction

Mech's programming model incorporates two distinct feedback loops to cater to different aspects of the programming process: the runtime loop and the live coding loop.

4.1.1 Feedback Loops. The runtime loop manages the program's execution cycle, automatically reacting to incoming data from various sources and updating dependent blocks accordingly. This efficient feedback loop reduces the amount of work required from users when specifying systems, streamlining the handling of data-driven systems.

The live coding loop focuses on the development process, providing a responsive and interactive environment for users to iterate on their ideas and refine their programs. Users can see the immediate results of their code changes, fostering a

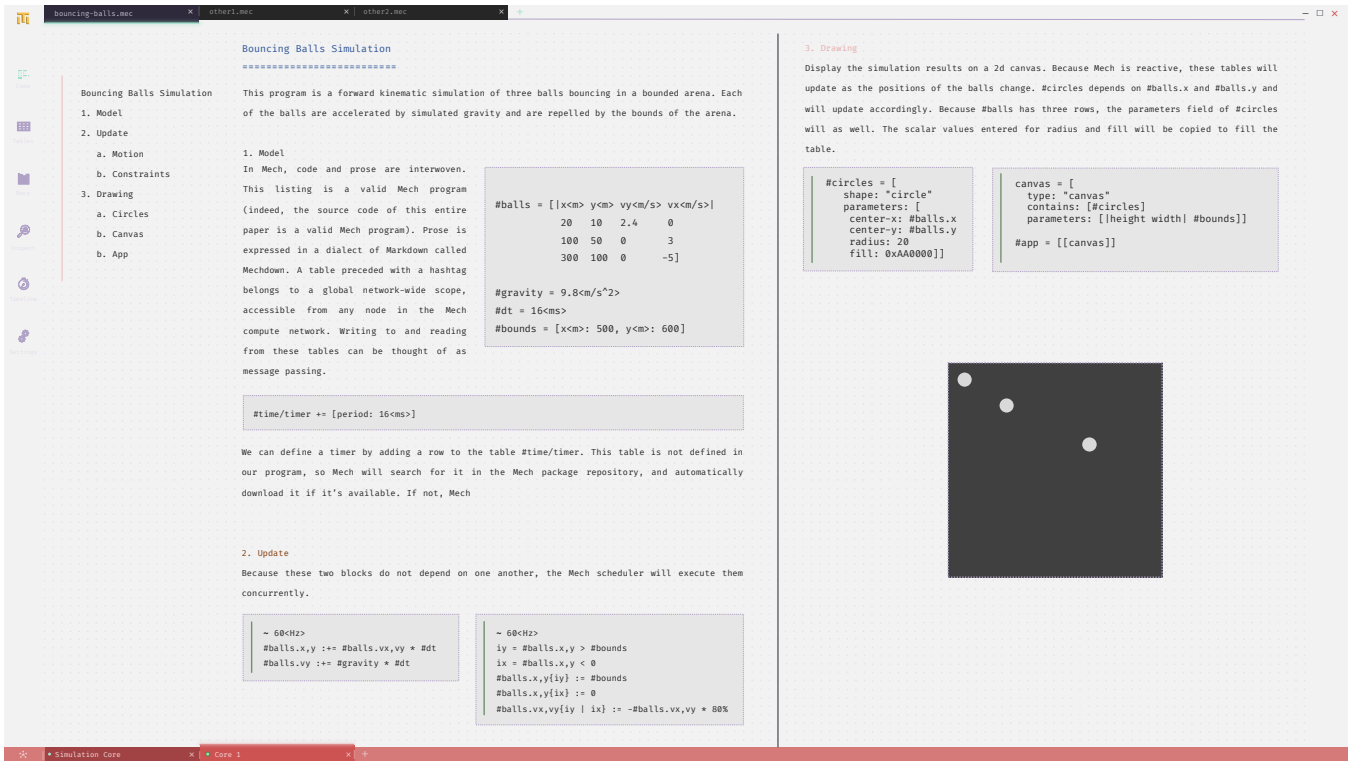

Figure 4. Design mockup of a prototype Mech IDE, written in Mech. Features tabs to manage editor panes (top), tabs to manage cores (bottom), tabs for opening and closing side panels and toolbars (left). The center area features a canvas, where developers can add rich text, Mech code, and can even visualize program output.

more efficient development experience. This real-time feedback also enables debugging of running systems, allowing users to diagnose and fix issues without time-consuming restarts or code rebuilds.

4.1.2 Abstraction Construction.

Abstraction construction in Mech is centered around blocks, which facilitate building and refining programs incrementally. Users can create abstractions by defining new blocks while the program is running, and seamlessly integrate them into the existing system. This incremental approach offers several benefits, including real-time feedback for evaluating the impact of changes and aiding debugging and optimization. Users can maintain a high level of confidence in their program's correctness and performance as they build up abstractions incrementally.

Figure 4 depicts the Mech IDE, built using Mech itself, provides a flexible and user-friendly development environment. It features a tabbed interface for managing editor panes and cores, customizable side panels, and a versatile canvas that combines text, code, and visualizations. Additionally, its real-time adaptability allows for on-the-fly adjustments and ad hoc tooling support. This rich set of features streamlines the development process and enhances the programming experience, making it easier to create efficient and adaptable Mech programs.

4.2 Notation

4.2.1 Notational Structure.

Mech uses a textual programming notation that is designed for clarity and conciseness. This structure is built around global tables, user-defined functions, custom data types, and array-based operations, allowing for a clean and organized representation of robot programming tasks. The notational structure in Mech is tailored for expressing complex behaviors and managing interactions between various components of a robotic system.

4.3 Surface and Intentional Notation

Mech's notational structure is built around global tables, user-defined functions, custom data types, and array-based operations, which allow for a clean and organized representation of robot programming tasks. Tables are a crucial element of Mech's notational structure, serving as a universal data type and allowing for the representation of complex data structures in a simple and intuitive manner. By relying on tables, Mech avoids the need for complex data conversion and a proliferation of symbols in the notation. Instead, tables provide a consistent and uniform way to manage and manipulate data.

```
1  Mech Integrated Development Environment (IDE)
2  ================================================
3
4  When the compile button is clicked, take the contents of the code area and compile it.
5    ~#compile-button
6    #notebook/compiler := #code
7
8  Add a gui application
9  #app = [[#grid]; [#bottom-panel]; [#left-panel]; [#app-area]]
10
11 Render the app to the screen.
12   #app-area = [
13   kind: "panel-center"
14   contains: [[#tables-panel-contents];  [#docs-contents]; [#top-panel]; [#app-contents]]
15   parameters: [margin: 0, padding: 0]
16   ]
17
19 Draw a circle around the cursor
20   #cursor = [
21     kind: "canvas"
22     contains: [
23       shape: "circle"
24       parameters: [
25         center-x: #io/pointer.x
26         center-y: #io/pointer.y
27         radius: 25,
28         fill: 0x3E3848]]]
```

Figure 5. Code selections from the Mech IDE. Lines 4-6 demonstrate reacting to a button press by sending text to the Mech compiler for compilation. Lines 9-16 shows the layout of the application; its structure is a table of tables. Line 19-28 shows how the IDE can react to the user moving the mouse.

4.4 Primary and Secondary Notations

The primary notation in Mech is the textual programming language, which includes all essential constructs for defining robot behaviors and interactions. While the language does not rely on secondary notations, it is compatible with visualizations and other tools that can enhance understanding and provide additional context for users. These secondary notations can be used to complement the primary notation when necessary, but the primary notation remains the core means of expressing programs in Mech.

4.5 Expression Geography

In Mech, similar expressions are designed to encode similar programs. The language promotes consistency and uniformity in its notation, making it easier for users to recognize patterns and understand the relationships between different parts of the code. This expression geography is an important aspect of Mech's design, as it aids in readability and comprehension of the programs.

4.6 Uniformity of Notations

Mech strives for uniformity in its notations by utilizing a small number of basic concepts that can be combined and extended to handle a wide range of robot programming tasks. This uniformity simplifies learning and using the language, as users only need to become familiar with a limited set of core constructs to create complex programs. The language has no keywords, and only a small number of operators. Square brackets are used exclusively for table declarations; curly braces are used exclusively for table indexing; and angle brackets are used exclusively for kind annotations.

5 Conceptual Structure

5.1 Conceptual Integrity

Mech balances conceptual integrity and openness in its design, offering a language that is both elegantly designed and highly adaptable to a wide range of use cases. At its core, Mech is built on a database that holds tables, which serve as a universal data structure that can represent anything from sensor readings to program logic. This uniformity promotes conceptual integrity by providing a consistent and organized way to manage and manipulate data, simplifying the programming experience and enabling users to more easily construct meaning.

5.2 Composability

Mech components are composable from the top to bottom. At the syntax level, values of any kind can be composed into tables. Multiple tables compose to form the basis of a block, along with transformations defined over the tables. Blocks compose on the basis of their data dependencies, and define a core compute network.

Cores themselves are made of many blocks, and themselves compose on the basis of their data dependencies, but also on their access controls; new cores will reach out to the local Mech network to find other cores, and will negotiate data depencies with them. If a remote core has data the new core needs, and the new core has a capability to access it, then the new core will download that data and integrate it into its own memory.

At the highest level, machines can compose as well. Machines are made of one ore more cores, and can be made of machines as well. This uniformly modular approach to composability allows Mech programs to be easily extended and adapted to a wide range of robotic systems and applications.

5.3 Convenience

Mech provides convenience to users by incorporating a rich set of built-in functions and features specifically designed for robotics applications. This prevents users from having to reinvent the wheel for common tasks, such as handling asynchronous communication, working with physical units, or managing parallel computations.

6 Customizability

6.1 Staging

Mech allows for both customization of running programs and inert ones. Changes made to the code can be applied dynamically, with running programs being able to adapt to modifications without requiring a complete restart. The changes made persist beyond termination, allowing users to build on and modify their programs incrementally.

6.2 Addressing and Externalizability

Mech provides a high degree of addressing and externalizability. Portions of the system's state can be easily referenced and transferred between different parts of the program through global tables. This promotes modularity and facilitates the integration of new expressions and modules that can alter the system's behavior. Furthermore, Mech's asynchronous and distributed nature enables seamless interaction with external systems, making it adaptable to various hardware and software environments.

6.3 Self-sustainability

Mech is designed to be self-sustainable, allowing users to modify and extend the system's behavior from within the language itself. With support for user-defined functions, custom data types, and dynamic dispatch, users can create new abstractions and extend existing ones as needed. Additionally, Mech's built-in safety features, such as integrity constraints, and its capability permission system, help ensure the stability and security of the system as it evolves; if a developer wants to lock down the system to prevent it from self-modification, they are able to craft a capability to express whatever access controls they prefer.

7 Complexity

7.1 Factoring of Complexity

Mech effectively handles the factoring of complexity by providing reusable components and abstractions that hide intricate programming details. Functions, blocks, cores, and machines are all used to encapsulate code in Mech, allows the user to decide the appropriate granularity of abstraction for their project.

7.2 Level of Automation

Mech provides a high level of automation, reducing the need for users to explicitly specify certain aspects of program logic. For instance, Mech's inherent asynchronous and distributed nature allows for automatic parallelization of tasks, eliminating the need for manual thread management or synchronization. Additionally, Mech's dynamic dispatch and built-in support for units (quantities) help automate tasks like type checking and unit conversion. These automated features not only simplify the development process but also reduce the likelihood of errors related to manual handling. Mech's reactive runtime frees developers from having to decide when and how to process events, and the distributed runtime makes the decision of when and how to ship data to remote cores effortless.

8 Errors

8.1 Error detection

Mech incorporates error detection mechanisms throughout development stages, from initial coding to program runtime execution. A key feature is type checking during compilation, ensuring consistency in variable types, dimensions, and appropriate unit conversions. This prevents severe errors, like those in the Mars Climate Orbiter incident, which resulted from unit discrepancies [24]. The system enhances code safety by preemptively detecting potential inconsistencies before they evolve into runtime errors.

Mech's runtime error detection and debugging are enhanced by its deterministic and idempotent database system. Determinism guarantees consistent output for the same input, simplifying debugging. Idempotence allows for 'time-travel debugging,' where developers can retrace steps to locate and understand errors. Additionally, replaying past events facilitates post-mortem debugging, enabling error analysis even after occurrence. These features together make Mech a reliable platform for writing safe, consistent code.

For instance, in a robot scenario, if an unexpected behavior arises, developers can replay the sensor data log during the error occurrence in Mech. Developers can then step through the Mech code as it processes this data sequentially, inspecting variables and states at each step. This not only pinpoints

when the error happened but also provides crucial insights into what triggered the unexpected behavior, offering a path to resolution.

8.2 Error response

Upon detecting an error, Mech responds in different ways, depending on the nature and severity of the error. For instance, when the system detects a unit conversion error during compilation, it alerts the user with an appropriate error message, allowing them to fix the issue before execution. In the case of runtime errors, such as violations of integrity constraints, Mech's internal database rejects the offending transaction, and the robot's state is reset to where it was before the transaction occurred. This ensures that the system remains stable and predictable even when errors occur.

Rolling back to a prior state when errors arise maintains system integrity and predictability. This strategy ensures data consistency by rejecting transactions that could violate system integrity, thereby keeping the system in a valid state. It also promotes system stability by reverting to a known state, minimizing the impact of erroneous transactions. This rollback also helps isolate errors for easier debugging, and sustains predictable system behavior by avoiding unpredictable states. Despite the associated costs, such as potential progress loss, the benefit of maintaining a reliable and predictable system outweighs these, especially in critical applications like robotics. Unfortunately this error handling scheme does not prevent a system from starting at an invalid state, in which case performance would be unpredictable.

9 Adoptability

Mech is designed to be accessible to a wide range of users, from experienced programmers to domain experts with limited coding experience. This requires careful attention to usability and learnability, as well as a focus on addressing social issues that can limit participation in technical fields. In this section, we will discuss how Mech approaches these challenges and strives to create a programming environment that is welcoming and supportive for all users. We will begin by examining Mech's learning curve and the tools and features that make it easy for users to get started and build competence quickly. We will then move on to discuss Mech's efforts to promote diversity and inclusion in the tech community, and how these efforts are reflected in the design of the language and its supporting ecosystem.

9.1 Learnability

Mech addresses the learning curve by providing a range of tools and features. One such tool is the advanced debugger, which provides logs and time travel debugging. The debugger helps programmers quickly identify and fix errors, which is particularly useful for beginners who may struggle to understand what is happening in their code. In addition, Mech provides helpful error messages that guide programmers towards possible solutions, reducing the frustration and time spent on debugging.

Another feature that supports the learning curve in Mech is live programming. With live programming, programmers can see the effects of their code changes immediately, which helps them understand the relationships between different parts of their program. This feature is particularly useful for beginners who may have difficulty visualizing the behavior of their code. By being able to see the results of their changes in real-time, programmers can gain a deeper understanding of how the code works.

Moreover, Mech's language-agnostic design promotes universal accessibility in programming education. It addresses the linguistic and cultural barriers often faced by non-native English speakers learning to program, as found in several studies [10] [11] [7]. These learners frequently grapple with understanding instructional materials, technical communication, and the simultaneous learning of English and programming.

Mech alleviates these issues by reducing reliance on language-specific keywords, inspired by the approach of Hedy [12], a programming language designed for novices that introduces syntax gradually, although Mech takes the idea to an extreme. This approach eases the learning curve and allows learners to focus more on the logic and concepts of programming, making Mech an inclusive programming platform for a diverse global audience.

9.2 Sociability

Mech is designed to be accessible to a wide variety of users, from beginners to advanced programmers. In addition to providing a gentle learning curve, Mech also tries to addresses social issues in computing and the world at large. By empowering people to use Mech, the language can serve as an educational tool to teach younger kids about robots and computers.

The outreach component of Mech, called Forward Robotics, aims to promote STEM education and has been successfully used in engineering outreach programs such as CHOICES at Lehigh. Over the Spring and Summer of 2022, Mech was used in an engineering outreach program called CHOICES, which aims to deliver STEM enrichment activities to middle school girls. Mech is intended to be used by students, so CHOCIES was an ideal preliminary environment for Mech.

In total, more than 60 girls representing twelve schools from the Lehigh Valley and surrounding areas attended and participated in the robotics activity. They used Mech principally to program LED displays on Sphero Bolt robot balls, using colored Emojis.

The students, who had no prior programming experience, created over two dozen such images, and were able to do so

quickly and with minimal help from instructors and undergraduate volunteers, who themselves had only basic training on the language. Although this was not a formal study of the Mech language, the results were encouraging and point to a need for future study of Mech with this demographic.

Mech is also free, open source, low-resource, and runs on many devices, including phones, making it accessible to students who lack access to computers. This means Mech could find its way into developing countries where low-power cellphone usage is higher.

10 System Evaluation

In this section, we evaluate the performance of Mech using two different programs, the bouncing ball example from earlier, as well as an Extended Kalman Filter implementation.

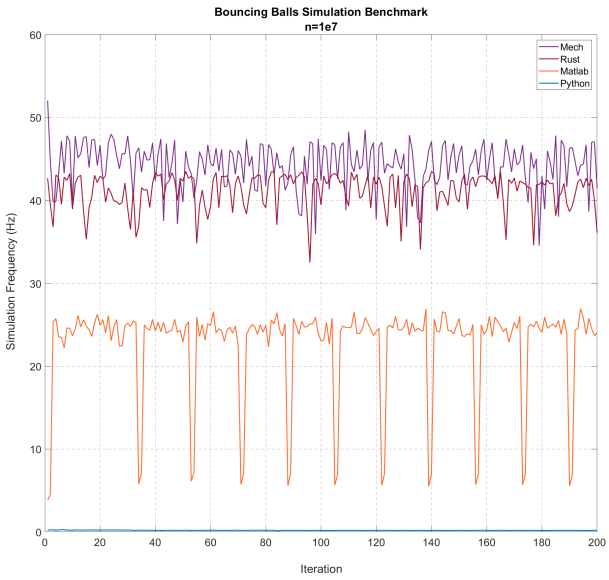

Figure 6. Performance of Mech (purple) compared to Rust (red), Matlab (orange), and Python (blue) on the bouncing balls simulation described in Listing 1, which is a CPU-bound task. All simulations were implemented natively in the respective languages. Simulations were run on a Quad-Core 11th gen. Intel Core H35 i7-11370H processor with 32GB LPDDR4x RAM. Fine-grained parallelism allows Mech to achieve top performance in this benchmark by fulling utilizing the CPU.

10.1 Bouncing Balls Performance

The bouncing balls program in Fix. 3 is a classic benchmark used to compare the performance of programming languages. It involves simulating the behavior of a set of bouncing balls within a confined space, with each ball bouncing off walls and other balls according to simple physics rules. This test is representative of Mech's workload, as it involves complex data manipulation and dynamic simulation, both of which are key features of the language. By measuring the performance of Mech on this test, we can gain insight into its ability to handle real-world tasks that involve data processing, simulation, and visualization.

We compared Mech against several other programming languages commonly used in scientific computing, including Python, Matlab, and Rust. The results, shown in Fig. 6, indicate that Mech outperformed Python by a significant margin, which is not surprising given Python's interpreted nature. Matlab exhibited behavior indicative of a garbage collector, which is not ideal for real-time simulations. Rust, which is known for its performance, performed at the top of the pack.

However, Mech managed to exceed Rust's performance by leveraging parallelism. This is made clear in Fig. **??**, which shows the CPU utilization of the respective languages over the test duration. The figure reveals that Mech was able to achieve 100% CPU aggregate utilization across all 8 cores, whereas the next highest utilization was Matlab at 78%. This is a significant achievement considering that Mech is a high-level language that abstracts away the complexities of parallel programming. This means that programmers can focus on writing code instead of worrying about parallelism, and still achieve impressive performance.

Note that Python's exceptionally poor performance is expected due to its dynamic nature and global interpreter lock. There are ways to improve performance of Python, through third party libraries, but this comparison used native code.

The results of the bouncing balls test demonstrate Mech's ability to compete with established scientific computing languages, while also providing a high-level programming environment with built-in parallelism. This makes Mech an attractive option for scientific computing, robotics, and other real-time applications where performance and ease of use are critical.

10.2 An Extended Kalman Filter Implementation

10.2.1 Implementation. Extended Kalman Filters are used extensively in robotics to perform state estimation. In this example, an EKF was deployed to estimate the state of a simulated robot in a 2D world. The robot can be observed by cameras, which report only the bearing of the robot in the camera's frame when it is within range. As the robot moves, its true position is updated, and the EKF algorithm uses the cameras' observations to update the estimated position. A description of this algorithm can be found in [28]. Relevant portions of the Mech implementation are shown in Fig. 8, which depicts the use of features such as matrix multiplication with the ** operator, user-defined functions, vector swizzling, the apostrophe transpose operator, asynchronous blocks, and operator broadcast over vectors. The algorithm as expressed in [28] is 22 lines, so the implementation here is about as compact as it can be while staying true to the

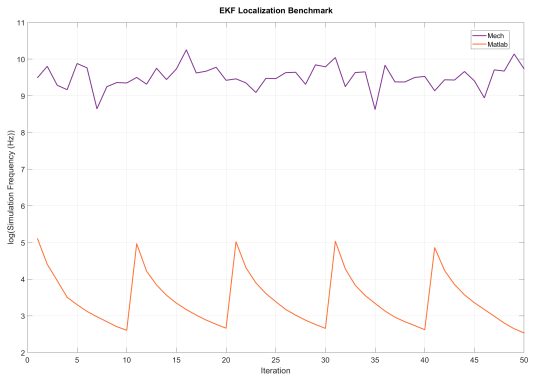

Figure 7. Mech (top) versus Matlab (bottom) performance on the EKF algorithm. This graph measures the simulation frequency of the EKF algorithm under both languages. The log scale on the y-axis means that Mech runs 500x faster than the Matlab implementation. The reason for this is likely due to how Matlab allocates memory within the simulation loop, whereas Mech pre-allocates and reuses pools.

source presentation. Notably though, when the Mech auto-differentiation engine is fully implemented, this example will be able to be simplified, as the Jacobians in the algorithm could be automatically derived.

10.2.2 Performance. The performance of Mech and Matlab was compared using the Extended Kalman Filter (EKF) algorithm, with the simulation frequency being the metric of comparison across a number of iterations. Fig. 7 shows that Mech significantly outperformed Matlab, running 500 times faster.

This difference in performance may be attributed to the way in which Matlab allocates memory within the simulation loop, as opposed to Mech's pre-allocation and reuse of memory pools. Matlab will allocate a new stack frame on each function call, because it doesn't expect that the function will be used after that context. Mech, however, does assume the function will be used again, because Mech programs are intended to be cyclic. Therefore, memory only has to be allocated once for the EKF filter.

Another source of optimization for Mech over Matlab leading to this result is that Mech's multiple dispatch compiler chooses the appropriate optimized function to use given the types and dimensions of the function arguments. All Matlab matrices can possibly change shape dynamically at runtime, so Matlab cannot make any optimizations here. Mech determines which tables are static and which tables can grow at runtime, and it can aggressively optimize for tables that do not grow, as is the case with the tables in the EKF filter, which has been shown to be beneficial by [6]

The comparison demonstrates Mech's efficient memory management and optimized execution, making it a compelling choice for high-performance computing tasks despite its simple syntax.

11 Discussion and Conclusions

In this paper, we introduced Mech, a new programming language and platform for robots. The combination of features offered by Mech may serve to achieve the goal increasing productivity and decreasing barriers to entry to robotics, and preliminary findings are promising in that regard. The implementation is open sourced and licensed under Apache 2.0 so that anyone can freely use and contribute to the language. Mech is still early in its development, but the language is nearing a beta release for public evaluation. Some of the features described in this paper, while they exist as a prototype in the system, are not fully implemented yet. Features which are working include: a distributed runtime, asynchronous execution, fine grained parrellism execution, performance, units and unit converstion (although not custom units), types tables, multiple dispatch on tables of different shapes, live coding environment with an editor and literate programming support, time travel debugging, native executables, a native GUI interface, a REPL, a testing framework and profiler, and the capability permission system. These features although implemented, are implemented but not complete, and may crash, or not be supported for all data types and systems. ROS integration is handled through a machine that leverages Rust's ROS bindings.

Currently in the experimental phase of development are GPGPU, automatic differentiation, AI integration (Stable Diffusion, ChatGPT).

The second quarter of the 21st century is going to witness the rapid advancement of robotic technologies and a contrast in hardware architecture from the 20th and early 21st century, where as many as 4 cores were available on a commercial-grade CPU, but certainly not hundreds or thousands. One way to make sure the robots work for the interests of every one of us to democratize access to the ability to program them. We believe Mech embodies the ideals of a platform that will help advance the field of robotics for everyone.

References

[1] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and Padmanabhan Pillai. 2007. Meld: A declarative approach to programming ensembles. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2794–2800. https://doi.org/10.1109/IROS.2007.4399480

[2] G. Biggs and B. Macdonald. 2003. A survey of robot programming systems. In *Proceedings of the Australasian Conference on Robotics and Automation*. 27.

[3] J. Bohren, T. Foote, J. Keller, A. Kushleyev, D. Lee, A. Stewart, P. Vernaza, J. Derenick, J. Spletzer, and B. Satterfield. 2009. *Little Ben: The Ben Franklin Racing Team's Entry in the 2007 DARPA Urban Challenge*. Springer, 231–255.

```
1  Extended Kalman Filter Localization
2  ===================================
3
4  [μ2, Σ2] = time-update(μ<m,m,rad>, u<m/s,rad/s>, Σ<f32>, Δt<s>)
5    θ =  μ.θ
6    Gt = [1  0 -u.v * math/sin(angle: θ) * Δt
7          0  1  u.v * math/cos(angle: θ) * Δt
8          0  0  1]
9    Vt = [math/cos(angle: θ) * Δt  0
10         math/sin(angle: θ) * Δt  0
11         0                        Δt]
12   μ2 = pose + u.v,v,ω * [math/cos(angle: θ), math/sin(angle: θ), 1] * Δt
13   Σ2 = Gt ** Σ ** Gt' + Vt ** Q ** Vt'
14
15 [μ2, Σ2] = measurement-update(μ<m,m,rad>, camera<m,m>, z<rad>, Σ<f32>, Q<f32>)
16   q = (camera.x - μ.x) ^ 2 + (camera.y - μ.y) ^ 2
17   Ẑ = math/pi2pi(angle: math/atan2(y: camera.y - μ.y, x: camera.x - μ.x) - pose.θ)
19   H = [(camera.y - μ.y) / q, -(camera.x - μ.x) / q, -1]
20   S = H ** Σ ** H' + Q
21   K = Σ ** H' ** matrix/inverse(table: S)
22   μ2 = (μ' + K * (z - Ẑ))'
23   Σ2 = ([1 0 0; 0 1 0; 0 0 1] - K ** H) ** Σ
24
25 Every time the robot moves, run the time-update step
26   ~ #robot
27   [#robot-estimate, #Σ] := time-update(μ: #robot-estimate, u: #control, #Σ, #Δt)
28
29 Every time a camera witnesses the robot, run the measurement update step
30   ~ #camera
31   [#robot-estimate, #Σ] := measurement-update(μ: #robot-estimate, #camera, #z, #Σ, #Q)
```

Figure 8. A partial listing of an EKF algorithm implemented in Mech. This listing demonstrates several features of Mech, including user-defined functions (lines 4-13 and 15-23), Unicode support in source code, native matrix multiplication through the ** operator, table transpose through the apostrophe operator (as in Matlab), vector swizzling, asynchronous blocks, and the use of various standard library functions.

[4] M. Buehler, K. Iagnemma, and S. Singh. 2010. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic.* Springer.

[5] M. Carroll, K. S. Namjoshi, and I. Segall. 2021. The resh programming language for multirobot orchestration. *CoRR* abs/2103.13921 (2021).

[6] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. 2010. Optimizing MATLAB through Just-In-Time Specialization. In *CC '10:*.

[7] Sayamindu Dasgupta and Benjamin Mako Hill. 2017. Learning to Code in Localized Programming Languages. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17).* ACM, New York, NY, USA, 33–39. https://doi.org/10.1145/3051457.3051464 Published:12 April 2017.

[8] T. Garnock-Jones and M. Felleisen. 2016. Coordinated Concurrent Programming in Syndicate. In *European Symposium on Programming Languages and Systems.* 310–336. https://doi.org/10.1007/978-3-662-49498-1_13

[9] G. C. Gini and M. L. Gini. 1982. Ada: A language for robot programming. *Computers in Industry* 3 (1982), 253–259.

[10] Philip Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. 1–14. https://doi.org/10.1145/3173574.3173970

[11] Carmen Nayeli Guzman, Anne Xu, and Adalbert Gerald Soosai Raj. 2021. Experiences of Non-Native English Speakers Learning Computer Science in a US University. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21).* ACM, New York, NY, USA, 633–639. https://doi.org/10.1145/3408877.3432437 Published:05 March 2021.

[12] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20).* ACM, New York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262 Published:07 August 2020.

[13] Jaguar. 2019. Jaguar Land Rover Finds the Teenagers Writing the Code for a Self-Driving Future. https://www.jaguarlandrover.com/news/2019/09/jaguar-land-rover-finds-teenagers-writing-code-self-driving-future

[14] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (2023), 13. https://doi.org/10.22152/programming-journal.org/2023/7/13

[15] T. Koolen and R. Deits. 2019. Julia for robotics: simulation and real-time control in a high-level programming language. In *2019 International Conference on Robotics and Automation (ICRA).* 604–611.

[16] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada. 2011. ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. *IEEE/ASME Transactions on Mechatronics* 16, 2 (Apr

2011), 321–329. https://doi.org/10.1109/TMECH.2010.2042722

[17] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (Jul 2018), 689–711. https://doi.org/10.1109/TSE.2018.2833109

[18] J. Marshall. 2021. The robot revolution is here: How it's changing jobs and businesses in Canada. *The Conversation* (Feb 2021). https://theconversation.com/the-robot-revolution-is-here-how-its-changing-jobs-and-businesses-in-canada-155267

[19] R. Mogk, L. Baumg"artner, G. Salvaneschi, B. Freisleben, and M. Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH*. https://doi.org/10.4230/lipics.ecoop.2018.1

[20] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. H"ahnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, and S. Thrun. 2009. *Junior: The Stanford entry in the urban challenge*. Springer, 91–123.

[21] S. Moros, L. Wood, B. Robins, K. Dautenhahn, and A. Castro-Gonzalez. 2020. Programming a humanoid robot with the scratch language. In *Robotics in Education*. 222–233.

[22] B. Mosely and P. Marks. 2006. Out of the Tar Pit. *Software Practice Advancement* (2006).

[23] F. Myter, C. Scholliers, and W. De Meuter. 2019. Distributed Reactive Programming for Reactive Distributed Systems. *The Art, Science, and Engineering of Programming* 3, 3 (Feb 2019), 5:1–5:52. https://doi.org/10.22152/programming-journal.org/2019/3/5

[24] NASA. 2019. Mars Climate Orbiter. https://solarsystem.nasa.gov/missions/mars-climate-orbiter/in-depth/

[25] S. Papert. 1980. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc.

[26] C. Pinciroli and G. Beltrame. 2016. Buzz: An extensible programming language for heterogeneous swarm robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Daejeon, South Korea, 3794–3800. https://doi.org/10.1109/IROS.2016.7759558

[27] K. Shibanai and T. Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. Boston, MA, USA, 13–22. https://doi.org/10.1145/3281366.3281370

[28] S. Thrun, W. Burgard, and D. Fox. 2005. *Probabilistic Robotics*. The MIT Press. 204 pages.