MARS: Reducing Hardware Integration Overhead in Robotic Systems through Language Design

Reilly Desai (RBE), Olivia Olsen (CS), Colin Masucci (CS) Advisor: Dr. Carlo Pinciroli (RBE/CS)

Abstract

Robotic systems development remains constrained by the complexity of hardware integration, where adapting software to new or reconfigured hardware introduces substantial overhead in configuration, validation, and iterative testing. Widely adopted middleware systems such as Robot Operating System (ROS) emphasize flexibility and dynamic composition, but defer most compatibility checks to runtime, limiting their ability to provide strong correctness guarantees during development.

We present MARS (Modular Abstraction for Robotics Systems), a domain-specific programming language that tackles these problems by elevating hardware abstraction to the language level.

Language vs. Library

We identified a need in robotics for a more robust approach to hardware management in software, aiming to both simplify hardware interaction and improve reliability and correctness. We determined that a language would be more effective.

Library	Language
☼ Familiarity, easier adoption	☼ Comprehensive error handling
☼ Compatible with existing tools and libraries	☼ Enforce abstraction
☼ Simpler development	☼ Domain-specific features
	☼ Greater research value

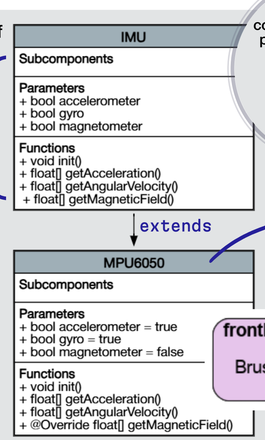
Component Configuration

In MARS, a robot is a tree of components, with the robot at its root and branching subcomponents.

Each component has a set of `subcomponents{}`, `parameters{}`, and `functions{}`.

We use an inheritance system so users can expect what information a component includes. Here, we expect every IMU to provide information on its axes.

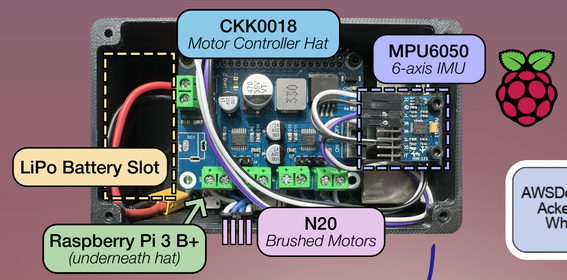
The interpreter's first step is to check the configuration. An error is thrown if, when a component is added to a robot, it does not implement all inherited fields.

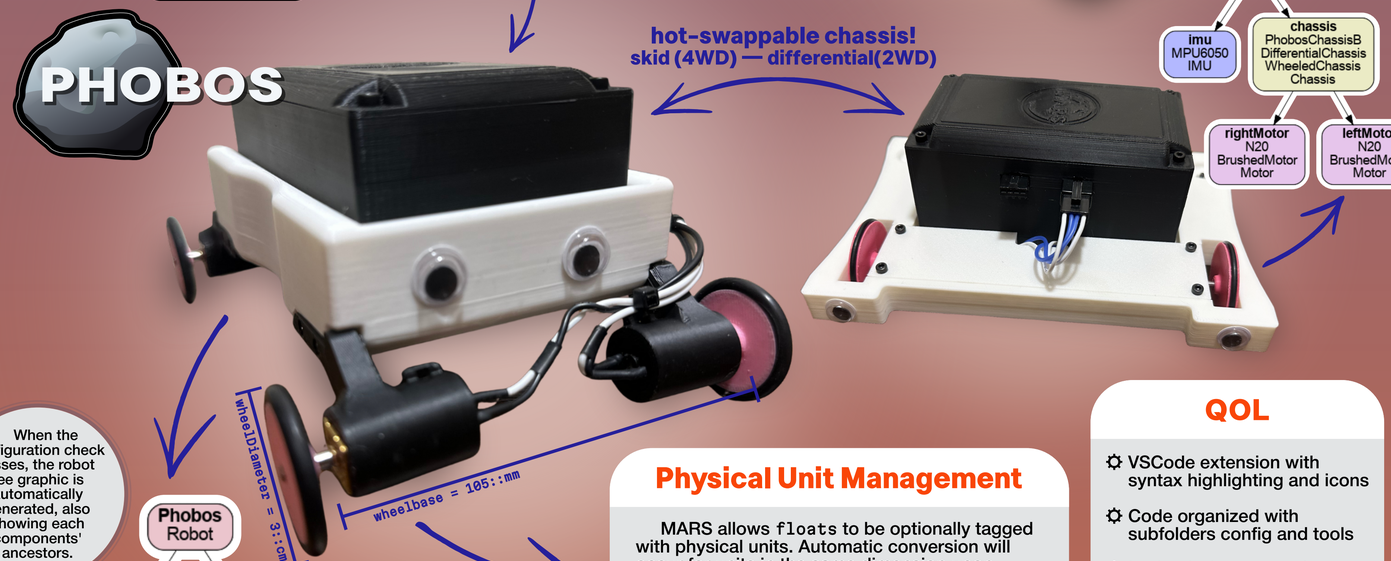


When the configuration check passes, the robot tree graphic is automatically generated, also showing each components' ancestors.

Experimental Evaluation

We evaluate MARS on two heterogeneous robotic platforms: Phobos and Deimos. Both robots are equipped with a wheeled chassis and a 6-axis IMU, enabling evaluation of ease of use when swapping hardware in MARS.





Acknowledgements. We would like to thank our advisor Dr. Carlo Pinciroli for his guidance and support throughout this project, Lab Manager Rachel Smith for her assistance with acquiring robots, and Casey Midgley for her assistance in fabricating Phobos.

Physical Unit Management

MARS allows `floats` to be optionally tagged with physical units. Automatic conversion will occur for units in the same dimension upon assignment and parameter passing. If conversion is not possible, an error will be thrown.

```
float::m meters = 100::cm;
print(meters, unit(meters));    → 1 m

void convertFt(float::ft length){
  print(length, unit(length));
}
convertFt(meters);  → 3.28084 ft
```

MARS will also automatically derive units from operations and assign units to variables accordingly.

```
float acceleration = 300::m / (25::s)^2;
print(acceleration, unit(acceleration));    → 0.48 m/s^2
```

Integration

If you can do it in Python, you can do it in MARS. Users can import any Python file into their MARS script, making it possible to integrate MARS with the entire Python ecosystem.

MARS also has a built-in ROS bridge that we use to interact with Deimos, an AWS DeepRacer that runs on ROS2-foxy. In MARS, a user might publish a message with:

```
publish("/cmd_vel", "geometry_msgs/msg/Twist", {
    "linear": {"x": 0.0, "y": 0.0, "z": 0.0},
    "angular": {"x": 0.0, "y": 0.0, "z": 0.0}
});
```

Subscribing has an extra layer to assure users can predict exactly when their data will update. Users can set component parameters to subscribe to a topic at compile time, and call `update()` to get the most recent message.

```
float vx = subscribe("/imu_pkg/data_raw",
    "sensor_msgs/msg/Imu/orientation/x");
```

Class Requirements

When users create a new class, they can designate a set of requirements that must be met by the robot that uses that class. Each requirement is treated as a conditional that must evaluate truthy.

The system is quite expressive, allowing for full control over components and their subcomponents, parameters, and functions with logical and comparison operators:

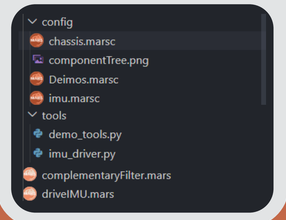
```
WheeledChassis;
  → robot tree must contain WheeledChassis

WheeledChassis(parameters=[wheelRadius > 1::in
        && wheelRadius < 4::in]);
  → chassis must have wheels with a radius between 1 in and 4 in

optional IMU(parameters=[magnetometer]);
  → recommends that the robot contains an IMU with a magnetometer
```

The `optional` tag allows a failed requirement to pass a flag instead of an error, permitting the interpreter to proceed to runtime. All errors and flags are returned at compile time, specifying the class, the robot, and failed requirement. Requirements are checked per robot each class is instantiated with, allowing for heterogeneous multi-robot support.

Crucially, this step is what ensures that users know their code is compatible with their robot before a single line executes.

QOL

- ☼ VSCode extension with syntax highlighting and icons
- ☼ Code organized with subfolders config and tools
- ☼ In-depth documentation using Docusaurus
- ☼ MARS will also be made entirely open source!



Future Work

- ☼ Expand the MARS component and class library, adding and demonstrating more common component structures, real-life robots, and standard robotics algorithms.
- ☼ How can we improve performance for high-demand robotics settings? Could a MARS compiler improve on the interpreter?
- ☼ Standardize distribution framework for integration with modern package managers (e.g. apt, aur, nix, pip).