MμSE: Supporting Exploration of Software-Hardware Interactions Through Examples

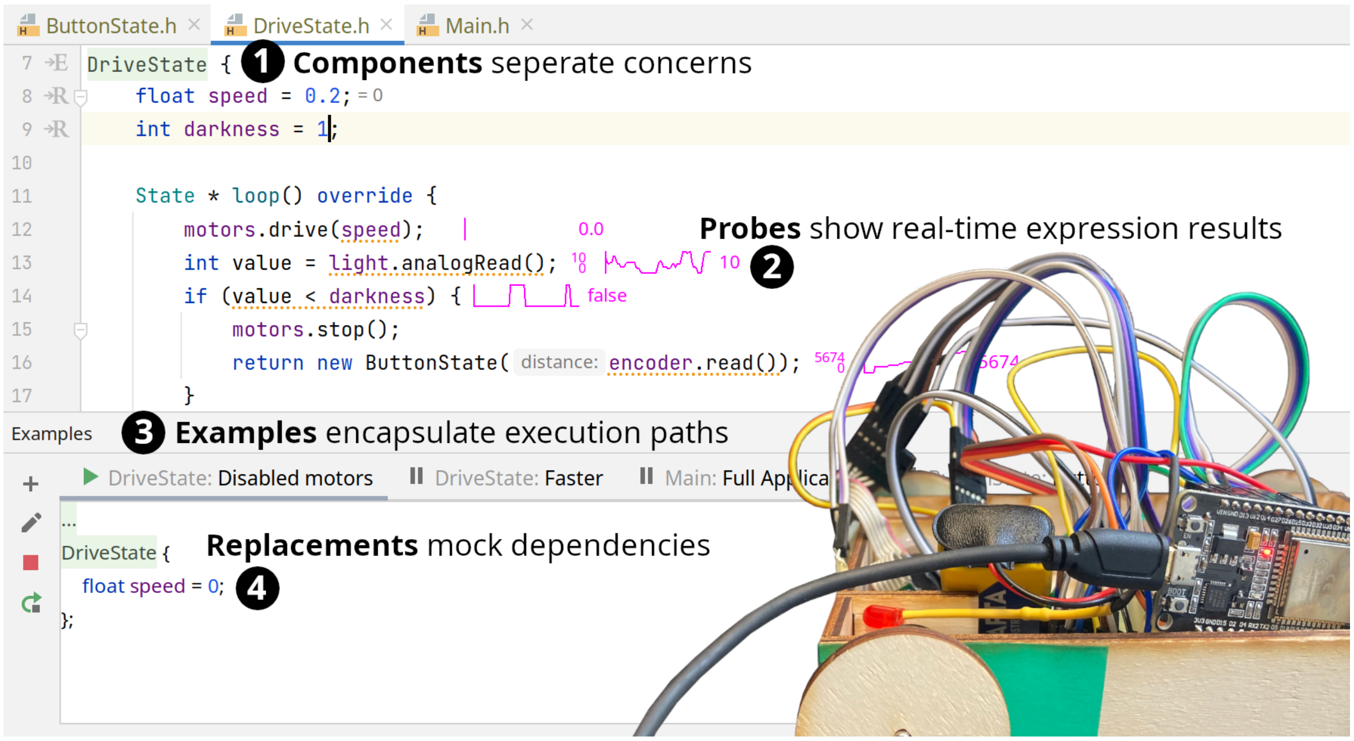
Paul Methfessel
paul.methfessel@student.hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Tom Beckmann
tom.beckmann@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Patrick Rein
patrick.rein@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Stefan Ramson
stefan.ramson@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany



ButtonState.h
DriveState.h
Main.h
1 **Components** seperate concerns
Probes show real-time expression results
0.0
10
2
false
5674
Examples
3 **Examples** encapsulate execution paths
DriveState: Disabled motors
DriveState: Faster
Main: Full Applica
Replacements mock dependencies
4

```
DriveState {
    float speed = 0.2; = 0
    int darkness = 1;

    State * loop() override {
        motors.drive(speed);
        int value = light.analogRead();
        if (value < darkness) {
            motors.stop();
            return new ButtonState( distance: encoder.read());
        }
```

```
...
DriveState {
  float speed = 0;
};
```

Figure 1: MμSE (Micro-Scenarios-for-Embedded) is an example-based live programming editor for prototyping embedded systems. It provides (1) components for testing and exploring hardware and software concerns in isolation from other implementation code, (2) insight into the runtime and external inputs with probes streamed live from the running hardware, and (3) examples for configuring live execution with (4) replacements of fields and methods.

ABSTRACT

Programmers regularly explore the execution of code examples to verify assumptions by adding print statements or commenting in and out setup code in their implementation to isolate code paths of interest. In our formative study on developing embedded programs, where proximity to hardware dictates low abstraction levels, we observed that wrong assumptions occur frequently. However, traditional editors for embedded programs lack support for such

CHI '24, May 11–16, 2024, Honolulu, HI, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0330-0/24/05
https://doi.org/10.1145/3613904.3642186

explorations. Consequently, programmers have to re-create and clean up setup and print statements in their code for each example.

MµSE supports isolated explorations of code examples by promoting examples to first-class entities that allow for the mocking of side effects from code and hardware, which could interfere with examples, and automatically showing values of expressions, replacing print statements for debugging. Our exploratory study found that MµSE supports participants in developing an understanding of software and hardware components and identifying false assumptions from observation of incorrect behavior.

CCS CONCEPTS

• **Software and its engineering** → **Application specific development environments**; *Integrated and visual development environments*; • **Computer systems organization** → *Embedded software.*

KEYWORDS

live programming, embedded systems, examples

ACM Reference Format:
Paul Methfessel, Tom Beckmann, Patrick Rein, Stefan Ramson, and Robert Hirschfeld. 2024. MµSE: Supporting Exploration of Software-Hardware Interactions Through Examples. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613904.3642186

1 INTRODUCTION

Embedded systems consist of a computer as a part of an electrical or mechanical system called a micro-controller unit (MCU). Writing code for these devices often requires gaining knowledge on and prototyping of these external systems. If, for example, a microcontroller should read values from a light sensor, the developer needs to know how to connect it correctly, calibrate it, and interpret the resulting values. Execution typically also depends on the state of the physical environment around the device, such as the position of the device or brightness in the environment. Consequently, when errors occur, their cause might be because of a previously untested environment, faulty code, unreliable electric components, or wrong connections [3]. On top of the abstraction levels introduced through code, debugging an embedded program thus requires developers to investigate multiple other sources of potential faults.

To better understand how developers currently deal with these challenges when prototyping embedded code, we performed a formative study with developers of embedded programs. We found that participants of our study would begin by thinking of concrete examples [6] with a fixed environment and inputs. These concrete examples serve to identify wrong assumptions or gaps in the developers' mental model with regard to the actual execution. Developers of general-purpose software follow the same process [6, 21]. Participants described working in a bottom-up fashion, first creating code for the individual hardware devices, and then moving to higher abstraction layers, until the whole application is composed. Participants try and test their code for each hardware device against a concrete example to verify its behavior. These example execution paths are commonly created through temporary setup code that provides specific inputs and dependencies, while other application code that interferes with the execution is commented out. Later, when participants notice a gap in their understanding or encounter an error they suspect might be related to a hardware device, they often re-create those examples, either from scratch or recovering them from commented-out code, which introduces significant friction to switching between different examples and the full application. To see the values of the runtime, participants write print statements that need to be removed again later on. Similarly, to understand how program outputs affect hardware, such as the value for the speed of motors, programmers repeatedly output values as part of their temporary test setup, wait for the program to be compiled and flashed on the MCU, and observe the hardware's behavior. We conclude that participants use ad-hoc examples as a means to identify gaps between their mental model and the actual execution of the program [15]. In practice, however, participants reported that their workflow is not well supported by traditional editors for embedded programming. The friction of managing and re-creating ad-hoc examples in code discourages a structured and well-isolated exploration of errors across hardware and software layers.

To support developers in verifying their mental model when working on embedded programs, we created an example-based live programming [23] system called MµSE. With our tool, developers can directly create first-class examples in the editor to execute a component in isolation and mock dependencies, independent of implementation code. Switching between examples or the implementation is as simple as a single click. Runtime values of relevant expressions are shown inside the editor as *probes*, annotations that report the values of expressions, to help verify assumptions of state observable in the program as a replacement for manual, temporary print statements. While working in MµSE, the application code is continuously executed on the developer's host machine that communicates with a real MCU whenever code reads or writes from hardware devices through a small remote-procedure-call kernel on the MCU. When developers save their code, execution restarts immediately to support exploration of the effects of program outputs on the hardware, removing the wait time for compiling and uploading code when developers want to preview changes.

We conducted an exploratory user study with 13 developers, as well as two experts in the field, to understand how developers validate their mental model using MµSE and how it impacts the developers' workflow. We found that MµSE is especially helpful when developers are trying to understand an unknown behavior or value of hardware devices. In these circumstances, MµSE provides fast discovery of both errors and wrong assumptions about functionality.

Contributions. This work contributes:

- a formative study finding how developers prototype embedded systems code,
- the design and implementation of MµSE, an example-based live programming system that solves the identified problems of managing and testing examples for embedded programming, and
- an evaluation of MµSE with a focus on developers' interactions.

MµSE is publicly available on Github[1] under the MIT license to enable replication, further research, and use of the tool.

2 BACKGROUND

Developer-tooling for embedded programming faces limitations that are unique to its domain. In this section, we will describe differences between embedded programming and general-purpose programming, in particular differences in execution models and state management.

2.1 Obtaining Input from Hardware Devices

Hardware input for embedded programs is typically obtained from a set of sensors connected to the microchip. Sensors are read either in a push or pull manner: in a push manner, interrupts trigger registered entry points. In a pull manner, the program repeatedly reads the current sensor values in a loop and acts on it (this is also called Super-Loop-Architecture[2]). Both are illustrated in Figure 2.

Interrupts trigger events based on certain conditions, for example when an electrical signal exceeds a constant threshold. Interrupts tend to be low-level and require a good understanding of the frameworks' API, memory management of the MCU, and external hardware behavior. Since interrupts can be called at any time in the program execution, developers need to be well aware of the flow of the program to reduce the risk of race conditions or other problems [13, 20].

When using the pull approach, sensor values are pulled within the context of the main loop, which may contain arbitrary code and can thus react to arbitrary combinations of sensor values. As values can be continuously accessed from sensors and thus printed, the hardware may be more discoverable for programmers. It also requires less knowledge of the framework's API, since developers do not need to learn the specific triggers for interrupts, but can write them themselves with generalized logic. The disadvantage of this approach compared to interrupts is that it is slower and therefore not feasible for some high-speed real-time code.

In many use cases, programs need to integrate obtained sensor values over time by reading a sensor on each iteration of the main loop. For example, since sensor values are typically raw or only processed lightly, they contain noise that needs to be filtered over time. Or, for calculating an exponential moving average over light sensor values, the program needs to continuously access the sensor and accumulate its values. Accordingly, values need to be stored outside the local function scope when accumulating, as can be seen in Figure 3. Thus, the manner of pull or push in which the hardware communicates with the program influences both the control flow of the code and state management. While our proposed design is applicable to both pull and push communication, push communication is not compatible with the hot-reloading mechanism of our reference implementation, as described in section 7.

```
void loop() {
  if (digitalRead(pin) == HIGH) {
    Serial.println("Button pressed");
  }
}
```

(a) Pull/loop implementation: in the main loop of the program, the electrical signal at the pin connected to the button is checked. If it is *HIGH* the button is pressed and *Button pressed* will be printed until it is released.

```
void setup() {
  attachInterrupt(
    pin, onButtonPress, HIGH);
}

void onButtonPress() {
  Serial.println("Button pressed");
}
```

(b) Push/interrupt-based implementation: the interrupt is registered in *setup* to call *onButtonPress* when the given pin is *HIGH*.

Figure 2: Comparison of two programs with (nearly) identical behavior using a (a) loop/pull and (b) interrupt/push implementation

2.2 Observing and Debugging State in Embedded Programs

As described before, state is often accumulated or integrated over multiple iterations. As a consequence, the input cannot be as easily mocked for testing, as not only do the initial parameters of a function need to be set, but events triggered by the hardware need to be replicated with correct timing when mocked. Unfortunately, traditional tools offer little support for this.

Similarly, observing a single function execution without the context of previous iterations is less useful to the developer as the concerns of relevance are not apparent from a snapshot of the program state, but emerge over time. Debuggers can only provide insight into a stopped single frame and not an evolving context [18]. They also might cause problems when external triggers are ignored during a break-point or hardware keeps executing with the now frozen state of pins, for example, motors that keep driving while the program is suspended. Writing unit tests also is harder than for conventional software, since replacing external triggers in code with correct timing is not trivial. Instead, adding print statements is commonly used by developers for debugging. Print logs allow viewing evolving values, with some tools like the Arduino IDE even formatting logs in a time-value graph.

2.3 Reloading Code on MCUs

Executing new code on an MCU is typically done by compiling a binary on a host computer and uploading it onto the MCU. Depending on the size of the binary, combined compile and upload times are typically in the range of a few seconds (in our experience from seven to twenty seconds). In particular, to support building an understanding of the effect of values on hardware outputs, our tool aims to allow quick experimentation with different values and thus requires frequent changes to code. Consequently, uploading

[1]https://github.com/Paulpanther/muse-embedded-live-programming
[2]https://en.wikibooks.org/wiki/Embedded_Systems/Super_Loop_Architecture

```
int N = 100;
float avg = 0;

void loop() {
  avg += avg * (N - 1) / N + (float) analogRead(light) /
     N;
}
```

Figure 3: The *loop* function calculates an exponential moving average. The *avg* field has to be stored outside the loop. To validate the behavior, *loop* has to be called over multiple iterations.

code to the chip from scratch for each change incurs a major break in attention [29].

To bridge this gap, remote-procedure-calls can be used to run the newly written code not on the MCU, but on the developer machine [24]. Here only relevant platform calls like `analogRead` and `digitalWrite` are sent and evaluated on the micro-controller, to emulate the behavior a full execution on the MCU might have. This has the advantage of being a general-purpose solution available for many brands of boards and firmware and also allows wireless communication over Bluetooth or WiFi. However, it does introduce latency to each platform call (in our implementation a typical program is around seven times slower). It can also hide errors when code might not be executable on MCUs because of memory constraints.

3 FORMATIVE STUDY

To understand the developers' workflow when solving prototyping tasks with embedded software, we conducted interviews with eight participants (CS students, all male, 1 bachelor's, 6 master's, and 1 PhD student) who had previous experience with building simple embedded systems using the Arduino platform. Through a semi-structured interview with the authors, the participants described aspects of their procedure and thought process when approaching embedded programming tasks.

The interviews took around 20 minutes each. Quotes from participants are labeled P1 through P8 and are translated from German. As part of the interview, we prompted the participants to describe how they would write code in a scenario where a robotic car outfitted with a color sensor should find and follow a line. The task was specifically designed to require an exploratory [27] approach from the participants, where questions about how hardware actually behaved remained open. The authors followed up with additional questions to clarify points and to better understand the participants' thought processes.

Throughout the interviews, a pattern emerged, where participants highlighted the importance of exploring and verifying the behavior of the hardware. Participants generally appeared cautious to trust that once working code would remain working, stating that hardware was notoriously prone to break in unexpected ways. Consequently, the workflow participants described begins bottom-up, close to the hardware, but keeps revisiting bottom layers, when mismatches between the participants' mental model and execution surfaced through incorrect behavior of the system at later stages. As concerns and errors are often interconnected, participants described isolating an effect or cause as important but challenging. Participants did not specifically mention documentation as a strategy to further their understanding. We assume that it would play an important role during the assembly of the robotic car but may not be as helpful when experimenting with the embedded program, as documentation commonly lacks examples that illustrate the use in code suitable to the concrete scenario that is required. In the following, we detail the insights gained on the participants' workflow.

3.1 Developers Decompose Code in Components

Throughout all interviews, participants described starting with individual components of the physical hardware, in the case of our scenario the color sensor and motors of the car. For example, P2 answered the question of what the first executable program is with: *"Turning the motor on and driving forwards and backward"*. The same participant then explained that his next steps are to *"encapsulate the interaction with hardware into functions"*. When all hardware devices are wrapped into components, participants stated to then write application code, also in separate components, that makes use of the components for the hardware devices.

The distinction in components was most often done via classes, some via files, and others only with separate functions, but shared global variables. However, some participants admitted that while they preferred distinction into components, in a real situation code might not be structured that much because they have had bad experiences with isolation of components with the Arduino IDE in the past.

3.2 Developers Seek To Isolate Example Execution

Participants suggested a distinction between implementation code that is written for the final task and temporary code that serves to construct examples. These examples serve to build an initial understanding of a component or investigate errors they suspect to be in a component; so, more generally examples serve to align the developer's mental model with the execution. In the quote above, P2 described driving the motor forwards and backward as an example to gain an understanding of the motor's behavior and value ranges for the speed it accepts. While this example executes the core functionality of the *Motor* component, which will later be used in the application, the code written for this specific example is a temporary artifact that participants clean up before moving on, either by deleting it or commenting it out.

Interestingly, when prompted about handling errors, participants described recreating examples, often in the same or a similar manner as previous examples they used to gain an initial understanding. A particular challenge of revisiting an example arises, as by now, the implementation code may have grown to intertwine with the originally small component, and thus presents a barrier to testing. As having many components may affect the concrete execution of the MCU, participants described efforts to isolate examples, for instance by commenting out other components and code, or placing new example code at the top of the main entry point. Once the error has been found and fixed or a different hypothesis needs to be tested, all the temporary changes need to be reverted once again.

3.3 Developers Observe Inputs and Outputs Differently

As described, examples are designed to align a developer's mental model with the execution. Once participants manage to reach the execution of the desired aspect, they need a means to inspect the response and draw conclusions for adapting their mental model according to the observed behavior. Depending on where the execution manifests visibly, they use different strategies to inspect it.

When possible, participants verify the correctness of their code by observing the running hardware. In this case, participants try giving the hardware device API's values or parameters that they suspect may cause interesting behavior and observe the effect on the hardware device. For example, when driving the motors, participants may provide large and small values for the speed parameter, as well as negative values, to see how the motors react. An important hurdle for this workflow is the time it takes to compile and update a new binary, as described in subsection 2.3, which may mean that the most time spent during the exploration of a hardware device is on waiting for the next execution. This breaks temporal immediacy and might mean that developers become distracted easily from their task [29].

If input into the program is concerned, participants stated that they use print statements to observe runtime values via the serial connection on their host computer's log output. For example, when starting to work on reading the color sensor, participants stated that they use print statements to understand the values of the sensor and its behavior in the current environmental conditions. Further, participants modify the environment by placing the robot, and thus the sensor, in different places, and observe the resulting values, to better understand its response. Any print statements that were introduced are also conceptually part of the example's test setup and will thus have to be cleaned up once participants return to work on the implementation code.

4 DESIGN

In section 3 we described how developers construct examples to validate their mental model. In their current workflow, developers use ad-hoc approaches to reach what we describe as components, isolated examples, and means of verification. Components can be achieved through the built-in abstraction mechanisms of the language but are sometimes perceived as overhead by participants of our formative study. Examples can be achieved through commenting out source code or reconstructing isolated execution paths, which incurs an overhead every time participants move between the execution of implementation and example code. Verification is performed through print statements or observing the hardware, which is inhibited by the cost to re-create print statements and long upload times, respectively. Our proposed design, MμSE, supports, rather than replaces, the existing workflow of developers in these regards.

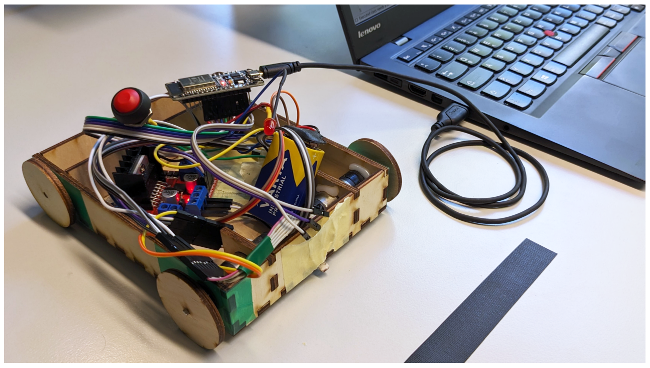


Figure 4: The setup of our developer in the example: the robotic car sits on the table, in front of the black line. It is connected via USB. The red button that needs to be pressed as part of the task is visible just atop the car. The light sensor is fixed to the car's front using tape.

MμSE's design consists of four parts. *Components* structure code in modules that programmers ideally design to serve a single purpose, such as controlling a motor, akin to well-designed classes in object-oriented programming. *Examples* are attached to a component and form new entrypoints for execution, in place of the global `setup` or `loop` functions. Examples thus allow starting execution directly at a component. Examples may contain *replacements* that override fields or methods of a component. When an example is executed, all replacements this example defines are first merged into the component code, potentially overriding component code. Together, component examples and replacements allow isolating execution to just a part of the system by moving the entrypoint and overriding code. By changing which example executes, developers can inspect different behaviors of the systems without changing their code. Finally, MμSE instrumentalizes the largest expressions of each line to introduce *probes*. These probes show a live preview of values that are seen for their expression during execution of the active example.

In the following, we describe a running example using a robotic car. Our developer wants to build an embedded program for the car that follows a simple procedure shown in Figure 5. In our example, the developer has the robotic car sitting on the table next to the laptop, connected via USB, as shown in Figure 4. In the following walkthrough, we will describe how the developer approaches formulating the code for the *Advance to Line* state using MμSE.

4.1 Components: System Modules

The developer starts the implementation with the interface for the low-level hardware devices, choosing the motor first. Here, the developer starts by creating a new *component* through the IDE's file menu. The developer names the component `Motor`. MμSE now generates a file for the new component which contains a basic skeleton stating its name as shown in Figure 6. In our reference implementation, a component in MμSE is a C++ struct that may have arbitrary methods and fields. While developers still have to choose a name and instantiate the new component later, MμSE

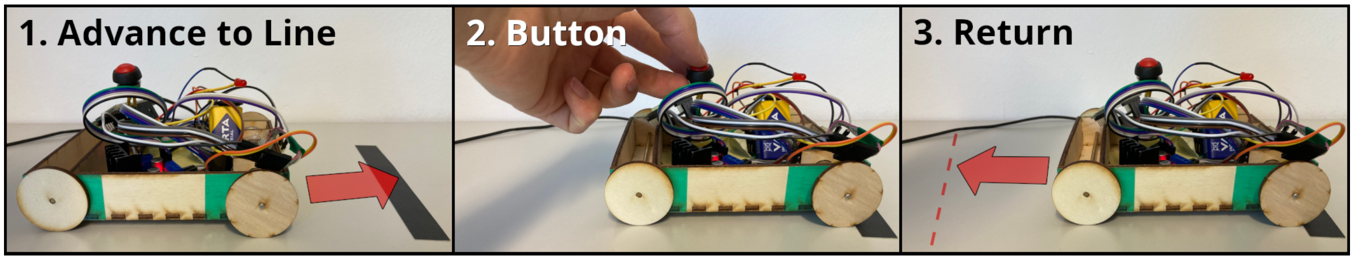
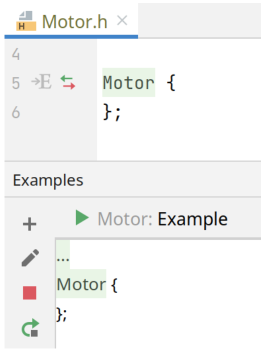


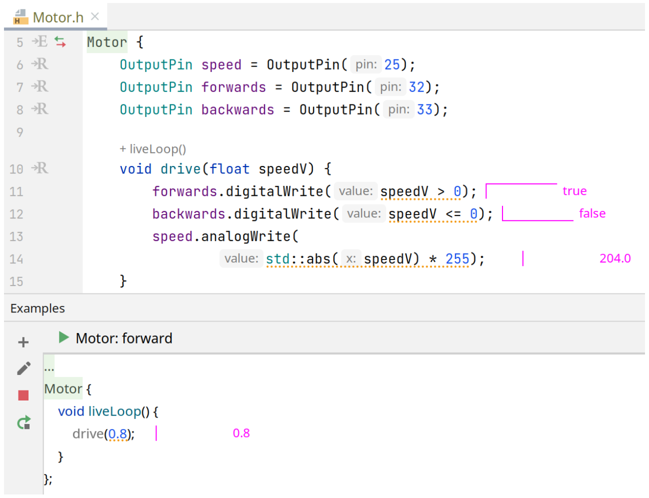
Figure 5: In the first state, *Advance to Line*, the car drives towards a black line, which the car detects using its light sensor. The car stops once the line is reached and transitions to the next state. In the *Button* state, the car waits until a button on the car is pressed and transitions to the next state. In the *Return* state, the car uses its encoder (a device to measure driven distance) to return to its initial position.



```
Motor {
};
```

```
...
Motor {
};
```

Figure 6: The generated skeleton of a new component named *Motor*. In the top code pane, the implementation of the component can be edited. To omit unnecessary implementation details from the developer, MµSE hides syntax that may not be changed by the developer, such as the `struct` keyword.



```
Motor {
    OutputPin speed = OutputPin( pin: 25);
    OutputPin forwards = OutputPin( pin: 32);
    OutputPin backwards = OutputPin( pin: 33);

    + liveLoop()
    void drive(float speedV) {
        forwards.digitalWrite( value: speedV > 0);      true
        backwards.digitalWrite( value: speedV <= 0);    false
        speed.analogWrite(
                value: std::abs( x: speedV) * 255);     204.0
    }
```

```
...
Motor {
  void liveLoop() {
    drive(0.8);      0.8
  }
};
```

Figure 7: The developer populated the *Motor* component (top) with three fields and a `drive` method. The example is renamed to *forward* and calls the component's `drive` function in the example's `liveLoop`. The green arrow indicates that this is the active example, meaning that `drive` will be invoked and the car is driving forward. Note that MµSE provides wrapped components for `analogWrite` (as `OutputPin`) and other Arduino API calls, such that users of MµSE can use the same conveniences for the API as with their own components.

removes the friction that usually comes with introducing abstraction to encourage developers to place separate concerns in separate components, even during prototyping.

4.2 Examples: Isolated Execution Paths

MµSE automatically creates an *example* for a new component. An example in MµSE consists of

- a `liveLoop` method that is continuously executed while the example is running, which is empty by default,
- its associated component that will be instantiated when the example executes,
- a name identifying the example,
- any number of *replacements* that override functionality that exists in the component, explained in subsection 4.4, and
- any number of *external replacements* that override the functionality of component dependencies.

The definition of an example is placed in a separate code editor underneath the component's code, with a tab for each example, as shown in Figure 8. One example is considered the *active example*, signified by a green arrow. The example code editor initially contains an empty skeleton of the component, as shown in Figure 7.

The example code is stored alongside the components. Before execution, MµSE merges the example code into the component code. The merging adds any definition from the example that is not present in the component and overrides any existing definition. Thereby, the merging allows the developer to add methods, or override members or code without polluting the component code.

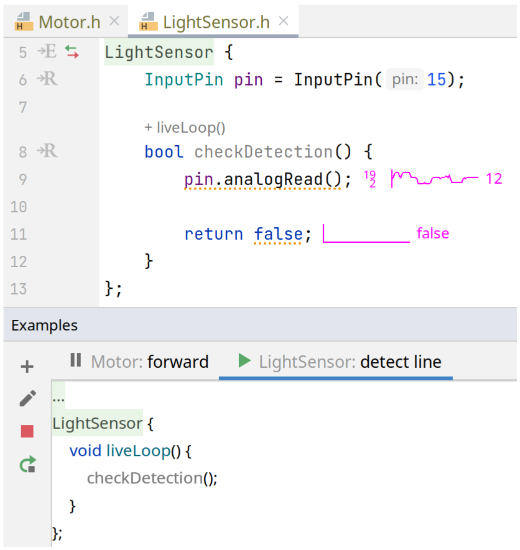
To start, our developer renames the default example to `forward` and sets it as the active example, as shown in Figure 7. Once the active example has been set, MµSE begins executing its `liveLoop` instead of the application's main execution entrypoint, thereby only executing code the developer explicitly asked to run within the context of that example. Whenever the developer saves changes in the code, the example will be restarted, meaning its previous component instance is discarded, a new one is instantiated, and `liveLoop` is executed again.

Next, our developer introduces a `drive` method in the `Motor` component that takes a decimal number denoting the speed at which the motor should drive. To observe its effect, the developer calls the `drive` method in the `liveLoop` of the *forward* example with an arbitrary number, choosing 5. The car now begins driving forward, at a very high speed. Aiming to reduce its speed, the developer chooses 1 instead but the car moves at the same pace. By changing the value to 0.5 and finally observing a slowdown, the developer concludes that values above 1 must be clamped.

Through further iterations, the developer finds a small speed that still overcomes friction at 0.2, which allows our developer to catch the car before it leaves the table. Before each new execution, if the car has moved too far, the developer has to manually reposition the physical location of the robot. To complete the *Motor* component, the developer then follows a similar process to create a `stop` method that sets the drive speed to 0.

Since the example code pane is separate from the implementation code pane but the example code is merged with the implementation code, developers can augment or adapt the execution of examples without affecting their final implementation code.

Now that our developer understands the motor device and has encapsulated its low-level workings in a component, the developer moves on to the light sensor that the car uses to detect when the line on the table has been reached. Our developer creates a new `LightSensor` component, creates a `checkDetection` method, names its example "detect line", and calls the new method in the example's `liveLoop`, as shown in Figure 8. To understand what threshold to use for detecting the black line, the developer writes `pin.analogRead()`, which reads the current sensor value.



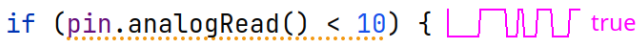
```
LightSensor {
    InputPin pin = InputPin( pin: 15);

    + liveLoop()
    bool checkDetection() {
        pin.analogRead(); 19 2 12

        return false; false
    }
};
```

```
...
LightSensor {
  void liveLoop() {
    checkDetection();
  }
};
```

Figure 8: The *LightSensor* component is executed by its *detect line* example. By moving the light sensor over the black line and away from it, the developer can see the read value in the probe. A sparkline shows the evolution of values, on the left side are the maximum and minimum observed values. The value on the right of the sparkline is the last reported value.



```
if (pin.analogRead() < 10) { true
```

Figure 9: Probes can also show boolean values. Here, MμSE automatically detected the full expression in the if-statement to be the point-of-interest and attached a probe next to it.

4.3 Probes: Automatic State Inspection

In MμSE, the outermost expression of each line of code is automatically detected as a *point-of-interest*. For each point-of-interest, MμSE inserts a *probe* [17] next to the expression, as also shown in Figure 8. The probe reports the current value of the expression, the minimum and maximum observed values, as well as a sparkline that visualizes recent values [8]. This proactive placement does allow the developer to see misconceptions about values, even when not specifically checking for them [14]. Consequently, developers may sometimes even spot problems while still writing the code, rather than spotting a problem only when observing the program's execution for correctness. For example, a probe inside a conditional branch that is never reached will not report values, and may thus indicate an incorrect condition.

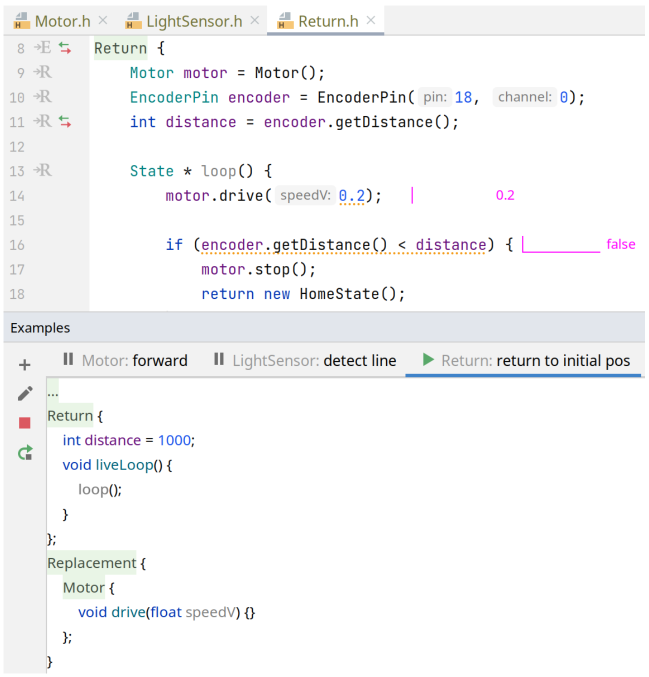
According to the point-of-interest heuristic, the `pin.analogRead()` expression for the light sensor receives a probe. Our developer can thus repeatedly move the physical light sensor over the black line on the table and observe the resulting sensor values. As the values vary heavily and refresh many times per second, the sparkline [28] and range of values help the developer understand the spectrum of values, which values are outliers, which values are typical, and how values are developing over time.

After choosing `10.0` as the threshold, the developer adapts the expression to `analogRead(pin) < 10`. MμSE now identifies the compound expression as the new point-of-interest and changes the probe to display binary values as seen in Figure 9. Our developer can now check that the probe shows `true` when the sensor is above the black line and `false` otherwise.

Now that both the components for the motor and the light sensor are completed, the developer creates a component for the *Advance to Line* state. Since the developer's uncertainties about the behavior of the hardware are eliminated and the components provide a higher level of abstraction, the developer can write the state's code without leaking details or assumptions about the hardware. Once the code for the state has been completed, the developer sets the example that was generated for the `AdvanceToLineState` component as active and observes that the car is behaving correctly.

As such, probes make print statements obsolete and even improve by not requiring the manual creation of statements and subsequent cleanup when they are no longer needed. Since values of sensor reads are always shown and updated continuously, probes reduce the gap between hardware and software and allow for a better understanding of internal hardware behavior.

4.4 Replacements: Stand-ins for Interfering Code



```
Return {
    Motor motor = Motor();
    EncoderPin encoder = EncoderPin( pin: 18,  channel: 0);
    int distance = encoder.getDistance();

    State * loop() {
        motor.drive( speedV: 0.2);   |          0.2

        if (encoder.getDistance() < distance) { |________  false
            motor.stop();
            return new HomeState();
```

```
...
Return {
  int distance = 1000;
  void liveLoop() {
    loop();
  }
};
Replacement {
  Motor {
    void drive(float speedV) {}
  };
}
```

Figure 10: The implementation and example for the *Return* state. In the example pane on the bottom, two replacements have been added: first, the distance field has been replaced with a constant value of 1000 units. Second, the developer added a replacement for an external dependency, the Motor component. Here, the drive method has been replaced with an empty implementation.

The developer continues this process for the button hardware device, the *Button* state, and then the *Return* state, creating a component for each. When working on the *Return* state, the developer notices that the state requires information on the distance that the car needs to drive back. This information is obtained from an encoder on the robotic car, a device that stores the distance that the car has moved. When executing the *Return* state in isolation, the car has not moved yet, so the distance the encoder reads is 0. To validate whether the written code is correct without having to move the car each time beforehand, the developer creates a *replacement* [21]. Replacements in MµSE allow overwriting fields and methods of components attached to examples by overriding their initializers or implementation and are similar to mocks in unit-tests [2]. Through the use of probes, the developer previously found that a value of `1000 units` is a typical distance for the current setup. Accordingly, the developer overrides the distance field of the *Return* state, which would usually read the value from the encoder, and instead assigns a constant `1000` units. Now, when testing the example for the `Return` state, the developer can observe the car correctly driving back the hard-coded distance without requiring prior setup.

In the same manner, the developer can create *external replacements* for fields and methods of dependent components of the component attached to the example. For our *Return* state dependent components are the *Motor* and *EncoderPin* components, as seen in Figure 10. If the movement of the car is not relevant to the example the developer is working on, disabling it is possible by adapting the dependent Motor component. To do so, the developer creates an *external replacement* for the Motor component's `drive` method that sets its implementation to be empty, thus effectively disabling car movement. This external replacement is also shown in Figure 10.

Replacements are the final piece to make isolation of example execution possible. Thanks to replacements, developers can create sophisticated examples that mock prior execution, which allows developers to obtain immediate feedback for code that depends on a specific state in hardware or software to have been reached. They further allow gaining some control over aspects of the physical world, as in the replacement of the motor, because of which manual resetting the car's position is not necessary.

4.5 Fast Restart and Example State

A development tool may support a feeling of immediacy in a temporal, spatial, or semantic sense [29]. As already described in subsection 4.2, temporal immediacy, that is a small temporal gap between change and observation of execution, supports our developer when exploring values for the motor's speed.

MµSE implements live programming by discarding example and component instances when the developer saves and instantly starting new instances of both. The gap between restarts is near imperceptible to the developer, thus implementing level 4 liveness [26]. When the old component is discarded and restarted, the state that had accumulated in the component is reset. As an interesting consequence, the state between the physical world and the program may diverge upon reset, as MµSE has no control over the physical world: if the car has already moved past the line, a restart of the example will not reset its physical position, even though a reset of the position may have been useful and intended as part of the example the developer would have liked to formulate.

In general, instantaneous restart is an essential aspect of programming tool design to allow developers to experiment with values and observe their effect [12]. If, instead, developers had to wait for a restart, some questions, such as what is a well-working value for speed for the Motor, would be significantly more time-consuming to answer.

5 IMPLEMENTATION AND CASE STUDY

We have realized MµSE's reference implementation as a plugin for the CLion IDE. To support instrumentalization of application code and fast reload, MµSE does not execute the application on the MCU. Instead, the application code is executed on the host computer. A minimal remote-procedure-call (RPC) kernel is flashed to the MCU that receives requested outputs from the host and sends input from the hardware to the host, as shown in Figure 12. Executing the application code on the host computer allows introducing code instrumentalization for probes, examples, and replacements without incurring a significant slowdown. However, RPC communication introduces a performance overhead with each I/O interaction, which we discuss in subsection 5.2.

To make an MCU platform, such as Arduino, available for use with MµSE, the API of the MCU has to be implemented as RPCs: all API calls are forwarded from the host to the MCU. Implementing a platform can be done incrementally: as support for a new platform is added, platform implementors may choose to only provide RPCs for calls of relevance to their current task and leave the rest as empty mocks. Our reference implementation features an implementation for Arduino that we used both in our user study and the below case study, mapping most of the GPIO calls and some other utility functions of the Arduino API[3] in 90 LOC (lines of code).

5.1 Case Study: TonUINO

To demonstrate MµSE's applicability to a real-world code base, we used it to inspect the open-source Arduino project *TonUINO*[4], a hardware music player using buttons and RFID for control. The project is of medium to small size (around 4000 LOC C++ without dependencies) but features interesting I/O code for interacting with the hardware buttons and RFID.

To make TonUINO executable in MµSE, we changed its build system from the PlatformIO tools to use CMake and thus our platform code instead of Arduino's. Otherwise, as TonUINO already split their code into classes, use with MµSE is straightforward: for example, the project has dedicated classes for its playback buttons, or a potentiometer for volume control. The code base already includes code that is well-suited for dedicated examples. For instance, the Button3x3 class includes a pre-processor directive to disable the actual component during compilation and instead output debug information.

When the user requests execution of the TonUINO application in the IDE, our CLion plugin instrumentalizes the source code by wrapping expressions for automatic probes, as well as replacing fields and methods. The plugin then compiles and launches the instrumentalized copy in a backend process on the host machine (Figure 12). When an I/O operation, such as an `analogRead`, is encountered during the execution of the application, the backend process invokes the corresponding RPC, causing the MCU to execute `analogRead` and respond to the host with a serialized result value. If that expression was wrapped in an automatic probe, the result value is then also sent to our plugin along with an identifier for the probe, before normal execution of the application resumes. When receiving a probed value, the plugin uses the provided identifier to find the responsible user interface representation of the probe and updates it to reflect the new value.

Using MµSE to Explore TonUINO. To show the usefulness of MµSE for more complex projects, we used it to explore TonUINO's code base. MµSE is designed to support developers' code understanding, which is relevant both when formulating code from scratch, as well as maintaining or exploring existing code bases. We pick a case where we want to form a better understanding of the code base beyond what the documentation offers. As an example, we observe that the `Commands` component triggers `none` commands when activated by buttons that should start playback, in addition to the playback command. We are unsure whether those `none` commands are correct here or not. To investigate their source, we first remove the manual step of pressing the button each time to observe the behavior: we create an *example* in the `Commands` component that *replaces* the button hardware, as shown in Figure 11, and simulates the button press. We proceed to inspect the *automatic probes* to find that the source of the `none` values is `getCommand`, which we find in turn receives them from a `getCommandRaw` call. In `getCommandRaw`, we realize that the application may be configured to support multiple sources of input, for example, different types of button hardware. If an input is not available, the source returns a `none` value as shown in the probe, so we conclude that the behavior is correct. MµSE thus supports our investigation of the TonUINO code base through multiple layers of abstraction (from buttons, to commands, to raw commands), allowing us to isolate the behavior we are interested in and using probes to quickly trace the origin of values.

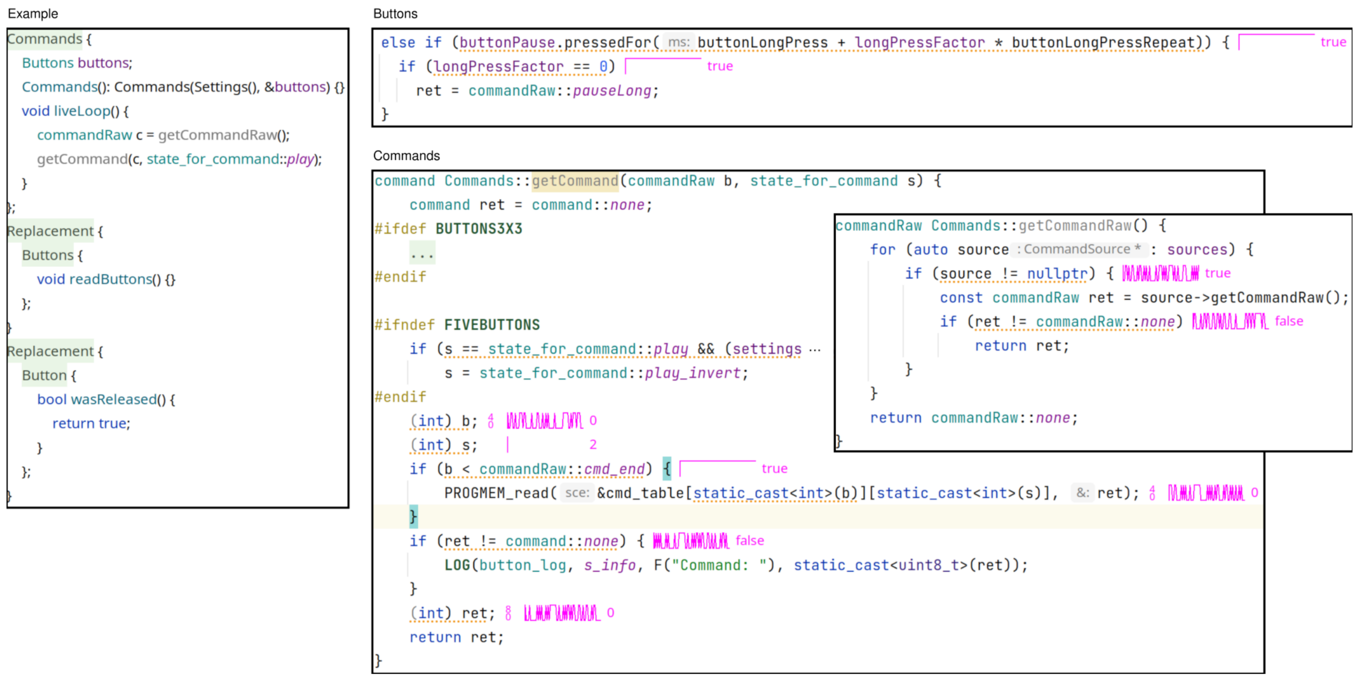
5.2 Latency Introduced by MµSE

MµSE prioritizes developer experience over runtime performance. This trade-off may render some applications unsuitable for our current implementation approach using RPC, depending on the use of I/O in the specific application, which our expert interviews confirmed (subsection 6.5).

To better quantify the performance loss, we took 1000 measurements for platform calls and averaged them. The connected microcontroller was an ESP32-WROOM32, the host computer runs on an Intel(R) Core(TM) i7-5600U 2.60GHz CPU. In that setup, for each platform call, the RPC communication through serial introduced an overhead of 5ms. Since non-platform-dependent code is run on the developer machine instead of the MCU, performance may differ here as well and might be faster or slower, depending on the host machine. In our setup, code on the developer machine was around five times slower than on the MCU.

On the other hand, as MµSE is launching the application on the host computer as opposed to uploading it to the MCU when changed, program reloads are faster. While compiling and uploading TonUINO via PlatformIO took around 12 seconds ($8.6s$ compilation, $3.4s$ upload) with our setup, MµSE took seven seconds to start the new version after a change ($1.5s$ instrumentation, $3.4s$ compilation, $1s$ startup). Both measurements were taken on subsequent builds where a build cache is already present. For our simpler user study project with the same configuration, we observed for MµSE restart times of 1.3 seconds ($0.4s$ instrumentation, $0.2s$ compilation, $0.7s$ startup), and for PlatformIO 15 seconds ($9s$ compilation, $6s$ upload, the high upload time appears to be due to less code optimization used by PlatformIO, leading to a four times bigger binary). Since compilation is required for PlatformIO and MµSE, execution times will grow with the size of the code base for both, however, MµSE could potentially benefit from C++ hot code reloading that is well-supported on computer platforms[5].

[3]https://www.arduino.cc/reference/en/
[4]https://github.com/tonuino/TonUINO-TNG
[5]See for example, https://learn.microsoft.com/en-us/visualstudio/debugger/hot-reload?view=vs-2022


Example

```
Commands {
  Buttons buttons;
  Commands(): Commands(Settings(), &buttons) {}
  void liveLoop() {
    commandRaw c = getCommandRaw();
    getCommand(c, state_for_command::play);
  }
};
Replacement {
  Buttons {
    void readButtons() {}
  };
}
Replacement {
  Button {
    bool wasReleased() {
      return true;
    }
  };
}
```

Buttons

```
else if (buttonPause.pressedFor( ms: buttonLongPress + longPressFactor * buttonLongPressRepeat)) {    true
  if (longPressFactor == 0)    true
    ret = commandRaw::pauseLong;
}
```

Commands

```
command Commands::getCommand(commandRaw b, state_for_command s) {
    command ret = command::none;
#ifdef BUTTONS3X3
    ...
#endif

#ifndef FIVEBUTTONS
    if (s == state_for_command::play && (settings …
        s = state_for_command::play_invert;
#endif
    (int) b;    0
    (int) s;    2
    if (b < commandRaw::cmd_end) {    true
        PROGMEM_read( sce: &cmd_table[static_cast<int>(b)][static_cast<int>(s)],  &: ret);    0
    }
    if (ret != command::none) {    false
        LOG(button_log, s_info, F("Command: "), static_cast<uint8_t>(ret));
    }
    (int) ret;    0
    return ret;
}
```

```
commandRaw Commands::getCommandRaw() {
    for (auto source :CommandSource * : sources) {
        if (source != nullptr) {    true
            const commandRaw ret = source->getCommandRaw();
            if (ret != commandRaw::none)    false
                return ret;
        }
    }
    return commandRaw::none;
}
```

Figure 11: Excerpts of the TonUINO project’s `Buttons` and `Commands` components, as well as an example that we added to the `Commands` component. The excerpts follow our example investigation: the example triggers the commands code for the play button but the probe at the bottom of the commands interface shows that the `ret` alternates between 0 and 8. Tracing the origin of the alternating values through the probes, we find `getCommandsRaw` that show the fluctuating behavior as some sources appear to be `null` according to the probes.

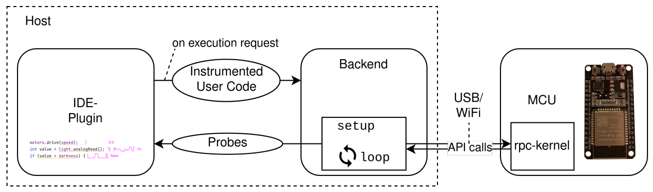


Figure 12: The execution model of MμSE is designed to improve reload times by omitting the expensive compile and upload process. Instead, when the developer in the IDE plugin requests execution, instrumented code is generated, and run in the system’s backend on the developer’s host machine. The backend captures API calls and sends these to the MCU and sends captured values from probes in the developer’s code to the IDE plugin.

6 EVALUATION

To understand how developers use our tool for building embedded programs we conducted an exploratory user study with the following research questions:

RQ1. How and when do developers address gaps and mismatches in their mental model while using MμSE?

RQ2. How does isolated, live execution of examples in MμSE impact the participants’ workflow?

6.1 Method

We conducted an exploratory user study with 12 CS students and one professional programmer (9 male, 4 female) with prior experience programming Arduino MCUs. The study was also performed with two experts; the results and findings from their group are discussed separately in subsection 6.5.

All participants were trained on using our tool for five minutes in a training exercise. After training, they were asked to solve a programming task. Finally, we conducted a semi-structured interview. Participants that we quote below are identified as P1 through P13, experts quotes as E1 and E2. All quotes are translated from German. Sessions lasted for 30 minutes each and participants were compensated 15€. During a session, both audio and the participants’ screen were recorded for later analysis.

In the main task, participants were given an assembled robotic car, with the same setup as in our running example in section 4, and a code template that already implemented a state machine and some low-level concerns. As in our running example, we then asked participants to use the motors, encoders, and light sensor to drive to a black line and store the driven distance. After a button press, the car should then drive back the same distance. The template is available as part of the supplementary materials.

We split the task into three sub-tasks, corresponding to the states of the state machine. Each sub-task was designed to pose questions similar to those that participants of our formative study had mentioned. Table 1 shows these questions for each state. All problems primarily concern the interface to the hardware, for example, valid value ranges or interpretation of behavior.

For the task design, we considered different sources of complexity [22]. A first version of the task, focused purely on comprehension, assisted by MµSE, but during pilot testing, we found that participants took too long to understand the code we had provided, which included some deliberate obfuscation to get participants to use the tool. The final version of the task could be completed by pilot testers and focused instead on a task that is created nearly from scratch, allowing participants to form a full mental model without prior input. Pilot testing also confirmed that the questions in Table 1 did occur.

6.2 Overview of Results

In the following, we describe the results of the test runs with the first group. We describe the results and insights from the expert test runs and interviews in subsection 6.5.

All participants were able to complete the tasks correctly within 20 minutes. Participants also all made use of examples and probes of MµSE to better understand the behavior of their code while working on the tasks. All participants implemented the sub-tasks in order of execution in the state machine. Four participants did not notice mistakes in a previous state before moving to the next state and had to go back to fix them. As in the pilot tests, participants did have to ponder the questions we designed into the tasks, as discussed in detail in subsection 6.3.

Two crashes of MµSE occurred during the test runs, the first due to a hardware fault in the provided robotic car and the second due to an untested edge case when executing code containing errors. Fortunately, both crashes occurred just after switching from training to the task, so the participants could begin after a restart of the tool and fixes to the hardware. While the low number of participants does not allow any quantitative analysis, we have collected qualitative insights from observations of participants' use of MµSE and interviews to answer our research questions below.

6.3 RQ1: How and when do developers address gaps and mismatches in their mental model while using MµSE?

MµSE is designed to reduce mismatches between the developer's mental model and the implementation's runtime. To classify different types of mismatches, we distinguish between known and unknown mismatches (or unknowns). The tasks in our study required participants to explore and understand at least the questions listed in Table 1, identified as Q1 through Q6. Sometimes participants were immediately aware of the unknowns present in the upcoming subtask, sometimes they only discovered the unknown when an error occurred.

In Table 2, we report how often participants treated assumptions as known unknowns (explicitly validating before writing code for the task) versus unknown unknowns (executing implementation code and seeking the wrong assumption). It was also noticeable that participants used different strategies for investigating unknowns. Some participants took more time to reflect on their knowledge and assumptions, to help discover unknowns to test before writing any implementation code. Others started writing code directly without reflecting on possible unknowns, seeking out the gaps in their understanding only once they became apparent through incorrect behavior during execution. While most challenges were binary in that they were treated as either unknown or known unknown by a participant, for Q6 we observed that three participants would only partially evaluate their assumptions before writing but later found additional unanticipated mistakes during the execution of the finished code. We include these instances in a separate *Both Unknowns* column in the table.

Investigating Known Unknowns of Input With Probes. Most participants explicitly investigated known unknowns of inputs, varying by question as shown in Table 2. By writing a read expression that, through MµSE's points-of-interest heuristic, receives a probe and by reloading the code, participants checked ranges, thresholds, or behavior of hardware inputs, as required in Q2, Q4, and Q6. For example, 11 participants found a suitable value for the light sensor threshold by writing `light.analogRead()` and observing the reported values in the probe when they held the sensor over the dark line or the white table respectively. Once explored, participants proceeded to construct code around their initial, exploratory expression. Many participants noted the helpfulness of probes while coding, for example, P5 said *"Trying out behavior and value-ranges leads to [...] me having more often the feeling that I understand what's happening"* and P7 noted *"I would have spent an order of magnitude more time [on the task] without the system"*. P5 also praised the placement of the probes on relevant code expressions: *"You can see directly where you need it what you are looking for"*. As MµSE places only one probe per line, we also observed that some developers would sometimes refactor a long expression into multiple expressions on separate lines when they wanted to view both as probes. While in the observed cases, code-readability tended to be improved by the involuntary refactor, four participants requested to be able to manually place multiple probes per line.

Investigating Unknown Unknowns of Input With Probes. Unlike known unknowns concerning input, where participants first wrote test code, for Q4, where a button signal has to be noticed, we observed that six participants first wrote the full code for the *Button* state, incorrectly assuming that the button returns `1` when pressed. Instead, those participants would execute the code for the finished state and only then notice their unknown unknown on the behavior of the button input. Here, MµSE's automatic probe placement was helpful, since it allowed participants to scan for mismatches in reported values with their expectations. For the Button state, we suspect that both the relatively lower complexity of the required code and participants' strong expectation that button-presses should return `1` contribute to the increased number of participants treating the assumption as an unknown unknown.

Investigating Known Unknowns of Output With Live Reload. Live reload was commonly used to verify known unknowns on the external state that is mutated through the MCU's output. The first action participants often took when starting the implementation of a new component was to write code that answered uncertainties on the correct approach to interface with hardware outputs. For example, all participants explicitly investigated the value for Q1, the slowest speed the motor can be driven at to still get the car to

Table 1: Questions in each sub-task, ordered by state. The code examples illustrate possible solutions in an abbreviated manner. The questions need to be resolved by participants to complete each sub-task successfully.

Id	State	Question	Code Example
Q1	Advance to Line	What is the slowest speed that still drives?	`motors.drive(0.2)`
Q2	Advance to Line	What is the best threshold to detect a black line?	`light.analogRead() < 3`
Q3	Advance to Line	How to stop the motor from driving when the task is done?	`motors.stop()`
Q4	Button	How does the button signal the pressed state?	`button.digitalRead() == 0`
Q5	Return	How to drive backward?	`motors.drive(-0.2)`
Q6	Return	How to detect a reached starting position?	`encoder.read() > distance`

Table 2: Number of times participants of the first group actively sought to answer a known unknown versus encountering wrong behavior during execution and seeking the corresponding unknown unknown per question. Sometimes we also observed that participants did not gain a full understanding of a question after actively attempting to answer a known unknown and therefore had to seek out mistakes that related to the same question. Correspondingly, these occurrences are reported in all three columns.

Id	Challenge	# Unknown Unknowns	# Known Unknowns	# Both Unknowns
Q1	`motor.drive(0.2)`	0	13	0
Q2	`light.analogRead() < 3`	2	11	0
Q3	`motors.stop()`	10	3	0
Q4	`button.digitalRead() == 0`	6	7	0
Q5	`motors.drive(-0.2)`	0	13	0
Q6	`encoder.read() > distance`	7	9	3
	Total	25	56	3

move forward. To do so, participants formulated a statement similar to `motors.drive(speed)` in the component's loop, trying different constants for `speed` and saving each time, to restart execution with the new constant. Similarly, Q5 was commonly solved by writing in code their assumption that driving backward can be done with negative speeds and testing it through execution. For both questions, the changed state is external, which means participants only had to observe the real world to validate the behavior. Instant reloading on save thus allowed participants to quickly explore value ranges for these outputs.

Lack of Support for Investigating Unknown Unknowns in Output. Almost all (10) participants struggled with Q3, where the motor had to be instructed to stop once reaching the line. While we assume that the root cause of participants' confusion was a quirk in the naming scheme of the API we provided to the participants, this revealed that MμSE provides little to no direct support for discovering unknown unknowns in hardware output. The `drive` method of the motor would have more aptly been called `setSpeed`, as it sets the speed value on the pin, which instructs the motor to drive until another value is given. However, these participants appeared to assume that omitting the `drive` call would prompt the car to stop.

6.4 RQ2: How does isolated, live execution of examples in MμSE impact the participants' workflow?

MμSE supports investigation of assumptions through isolation and live execution of examples, as described in subsection 6.3. However, both isolation and liveness may have implications for the developers' workflow; we will describe observations on both aspects in the following.

Participants Used Replacements To Accelerate Feedback Cycles. Replacements in MμSE allow mocking dependencies of components in examples. The *Return* state, where the robotic car's previously driven distance is required, was our task's most likely candidate for using replacements. Indeed, we observed that nine participants replaced the value with an arbitrary constant or a value seen before in state *Advance to Line.* All other participants who did not make use of replacements instead skipped testing the state and chose to test the entire state machine to see if their code for the *Return* state would work. Another use of replacements by two participants occurred when participants wanted to verify aspects of their components unrelated to driving in states *Advance to Line* and *Return* and thus created a replacement for the motor's speed to stop the car from moving.

Participants Struggled With Example Lifecycle. Our implementation initially did not include a means for developers to stop example execution, only to switch to a different example. Through feedback in our pilot for the study, we added a button to terminate examples manually, which would also reset (and stop) hardware devices through an annotated off-state for all relevant outputs, like running motors (these annotations were part of the code-template given to participants). Still, we observed a mismatch between the participants' workflow and the lifecycle of example execution during the task. The template that we prepared for the tasks modeled states as components that have their own `loop` method, which should return `nullptr` if the state machine is to remain in the same state or return the next state, otherwise. Accordingly, the examples for components modeled after states have a natural endpoint. The idea of an endpoint of examples conflicts with the example's `liveLoop`, which loops its body indefinitely while the example is active, to provide continuous feedback. While this was desirable and helpful when participants were investigating sensor values through the example, for state components the continuous `liveLoop` produced friction and confusion, as it was unclear to participants why their code kept executing, even though it should stop according to the state's lifecycle.

Participants Disagreed On Save-to-Restart's Benefit. As MμSE restarts the active example when the save shortcut (Cmd+s) is pressed, this presented an overloaded semantic to some developers. Two participants specifically mentioned that the combined save and restart action supports their workflow to obtain feedback. Others (six) pointed out that saving is a routine action for them that does not necessarily correspond to a finished thought or a valid program, as previously observed for live programming more generally [16]. Consequently, for these participants, restarts on save sometimes required them to react to sudden, unplanned changes in the hardware, as the example restarted its execution, which we will discuss in section 7.

6.5 Comparison Between Non-Expert and Expert Usage

To understand potential differences in the behavior of non-expert and expert developers, we also performed the study with two experts. These experts had high self-rated expertise working in embedded programming with MCUs (agree or strongly agree to the question "I have extensive experience or have received extensive formal education for programming for microcontrollers."). E1 reported to have 25 years of professional experience working with microcontrollers and E2 15 years. We followed the same test procedure for both the student and expert groups but included additional interview questions for the expert group concerning the use of MμSE in a professional context. We performed the expert tests and interviews remotely by giving the testers access to a host machine via remote desktop control and a camera feed for viewing the hardware. Upon request of the testers, we would interact with the hardware on their behalf as instructed (e.g., "move the car back", "press the button", "rotate the potentiometer").

For most of the user study, the experts showed the same assumptions as our first group. Known unknowns were observed for the motor's drive speed, the light sensors threshold, and the reversal of the motor's drive direction. Both experts directly wrote code to turn off the motor, which the majority of participants from the first group only thought to include after observing faulty behavior.

Both experts wrote large parts of the necessary implementation before executing the program. E1 mentioned that they may have felt discouraged to execute often by the remote testing setup but praised the quick restart times that they could otherwise observe. As the remote connection also slowed the progress down, we interrupted both tasks before the experts were able to fully complete them, in the interest of their time. We made sure to wait until after both experts had interacted with all parts of our system and had formulated code for all parts of the solution such that we could derive their assumptions before interrupting the task, such that a comparison between the groups is possible.

Usefulness of MμSE's Features. When asked during the interview, both experts agreed that having a real-time visualization of runtime values is helpful for their work, and E1 told us that *"[Probes] allow you to take a deep look into it [your program] and find your wrong assumptions; they give access to all the layers"*. Compared to existing tools, E1 mentioned that the Sparkline visualization excels at showing time-dependencies, which are often present in embedded development. Both experts remarked that most of their mistakes in their daily work come from small typing errors in complicated algorithms or simple wrong assumptions that will normally be overlooked but could be made visible through probes. When asked about the role of documentation, both experts mentioned that it is often incomplete or incorrect, or they form an incorrect interpretation on their first read, with E1 saying *"I prefer to rely on code if need be documentation inlined in code as comments"*. E1 further commented on the usefulness of replacements: *"This is exactly what I want, to be able to execute the business logic without dependencies"*. The experts recounted two use cases in a larger development setting which replacements would support: The testing of the software while the hardware is not finished yet (which may take multiple months), and supporting the simultaneous work of multiple software developers on the code, while only one hardware prototype exists. In both instances, the replacements would take the role of mocked components.

Challenges in Using MμSE in a Professional Environment. We further asked the experts about the challenges they see in using an example-based live-programming tool in their work. Both responded with the importance of a barrier-free workflow: there are classes of projects where real-time responses are important during the development as well, meaning latency cannot be introduced by the tooling. In addition, E2 mentioned that the automatic probes will likely be too distracting in large source files with multiple hundreds of expressions. Instead, E2 would prefer an opt-in mechanism for probes. E1 liked that MμSE is usable on existing code bases with few modifications and thus allows developers to collaborate on the same code base, independent of whether they use MμSE or not. In summary, E1 was excited about the prospect of using the concepts found in MμSE platform in a professional environment, while E2 expressed appreciation for the benefits of MμSE's features, especially for smaller code bases, but was skeptical of the runtime performance overhead and the possible distraction of always-on visualizations when applied to their own work.

7 DISCUSSION AND FUTURE WORK

MμSE is designed to align the developers' mental model with the execution of embedded programs, by helping them explicitly state and better verify examples. Our exploratory user study reveals that participants do feel supported and more efficient when working on embedded programs in MμSE. The study also reveals potentials and gaps in our design, along with observations of the preferred use of MμSE's features, which we will discuss below.

Idealized vs. Actual Use. Especially in the interaction with examples, we observed mismatches between the workflow that we designed based on input from participants of the formative study and the actual usage of participants in our exploratory user study. This includes

- not creating new examples for new use cases,
- not creating replacements for dependencies that hindered testing,
- mixing example setup and implementation code,
- not verifying assumptions before implementation code, and
- verifying multiple assumptions together and not separately.

We assume this stems from the participants' cost-benefit calculation [4], where the cost is the overhead of creating an example and replacements to ensure its isolation, and the benefit is an executable piece of code that documents an assumption and facilitates its later iteration, the example. When documentation or iteration of an assumption is deemed less important, for instance, because the participant already believed to have fully understood a concern, there is no need to create an example. Regardless, when the understanding does turn out to be incomplete, MμSE still benefits developers through automatic probe placement. As also observed in the study, automatic probes as a passive, always-active helper helped participants notice problems while they were formulating implementation code, even if they did not actively seek them out.

Generalized vs. Specialized API Interface. When designing MμSE, we tried to keep changes in the Arduino API and C++ syntax to a minimum so as to not confuse developers already familiar with the Arduino API and embedded programming. The concept of components as central language construct that structures the program, however, implies that using an existing code base with MμSE will require some refactoring to conform to the component structure. At the same time, as the Arduino API and other embedded programming frameworks are designed for low abstraction levels for communication with hardware, code written with these frameworks communicates little of the original intent programmers had until they introduce their own abstractions. Consequently, as MμSE only knows the low-level interface, MμSE misses out on the potential of more specialized visualizations or means to communicate and control state and behavior to developers. For example, the knowledge that a motor is installed in the robotic car could tell MμSE to display a probe near its usages in a component to show its current status (if it is driving and at which speed). Another possibility would be different probes for hardware devices, like color sensors and video cameras. There already are some systems that use knowledge over the domain to improve developer experience and even gain better performance in different metrics [1, 11].

Control-flow Visibility. At times, participants of our study sought to understand if certain parts of their implementation would be reached by control flow, for example, branches of an if-statement. Probes in MμSE already communicate aspects of control flow, for instance by annotating the result of a condition for an if-statement, or a moving or frozen probe graph indicating whether a line of code was reached. However, these solutions are indirect and could be improved, for example by deemphasizing lines that are not reached, as for example done in Babylonian Programming [21]. In our study, participants stated control-flow was not very understandable. Many participants for example made the mistake of confusing the example execution with the execution of the whole application, which could have been prevented if the currently executed functions were displayed more directly.

Per-Instance Replacements. Our design allows replacements of both the component attached to the example and external replacements of other components. However, all instances of this component are replaced. As a future extension, it could be possible to only replace one instance of a component or provide different replacement implementations per instance. For example, a developer might write a component that uses two light sensors for different purposes. To test the component's usage of the first sensor in isolation from the second, the developer might want to create a replacement for the second light sensor that only emits constant values.

Automatic vs. Manual Probe Placement. As described in section 6.3, participants of our study praised automatic probes. However, allowing the developer to manually place probes would be an important addition to ensure developers feel supported, not patronized, by the tool. There is also the possibility to extend the placement of probes to include more software-hardware boundaries, like showing current values on GPIO pins and where in code they are set, similar to Bifröst [18]. Additionally, users could define their own probe presentations to be used by MμSE.

Accessing Physical State. A major limitation of our solution is that it stops at the system boundary of the MCU. The physical state of the robotic car is not managed by MμSE and may thus break the developer's fast feedback loop. Future work could consider dedicated reset instructions that would, for example, drive the car back to a reset position. Alternatively, example execution could involve prompts to the developer to confirm that the physical state is arranged in a certain manner before proceeding, as is for example already done in the educational tool ElectroTutor [30]. These instructions could allow the description of well-defined scenarios that support reproducibility, especially when shared between different developers. Another problem is safety. P1 answered when questioned about the live response: *"I like it for non-dangerous components, like LEDs or buttons, for other things it is more difficult"*, pointing out that some hardware devices are quick to damage themselves or others when passed the wrong values. More control over physical state could prevent such accidents, by being aware of components needing more safety precautions and not activating them without warning. MμSE takes first steps in this direction, as annotations in program code inform the system that certain pins need to be reset when switching examples, such as setting the speed pin of the motor to `0`.

Limitations of the Reference Implementation. As already mentioned in the expert interviews in subsection 6.5, being able to use general embedded code without many modifications and a low-latency execution for real-time applications are both essential features for some classes of embedded programs. MμSE's reference implementation does not fulfill those requirements yet, as some concepts like interrupts (as explained in subsection 2.1) often used in embedded programming are not supported and performance overhead is introduced for platform calls, as discussed in subsection 5.2.

Both of these problems are a result of our RPC-based approach that splits execution between the host and MCU to benefit from accelerated restart times. An alternative design could compile and upload examples and implementation directly to the MCU, removing the need for an adapted implementation per platform, the communication overhead, and any prior limitations, such as interrupt support. However, a design that moves the entire execution to the MCU introduced overhead when communicating probe values and restarting the program. This impact of the overhead from probes would be significantly lessened, if probes would be re-designed to be opt-in, as requested by E2 (subsection 6.5).

Limitations of the Evaluation. We conducted an exploratory user study that sought to identify strengths and weaknesses in MµSE's design, testing with comparatively few participants and only for a small window of time. Similarly, the task we designed was kept small to fit in one continuous session and reflects only a slice of activities that developers engage in when working with microcontrollers. To assess how our observations generalize, a larger study would need to be designed that follows participants over a longer period of time.

8 RELATED WORK

The problem of mismatches in the developer's mental model and execution has been addressed by other tools, both for general-purpose programming and for the embedded systems domain. In this section, we want to compare existing solutions to our approach.

General-purpose Example-based Live Programming Tools. In our formative study, we saw that developers use ad-hoc examples to test and validate their mental model but that existing tooling does not fully support this workflow. We concluded that an example-based live programming tool would better support developers' behavior. For general-purpose software, this problem has already been encountered before. Babylonian [21] is another example-based live programming tool with the purpose of improving developers' understanding of their code. Both Babylonian and our tool let developers create explicit examples and view runtime values using probes. However, Babylonian focuses on local scope and annotates functions with examples instead of components. This is because, in general programming, functions are often the fundamental building blocks of programs and act as the central entry point for execution. Contrarily, because of the real-time architecture of embedded programs, with looping functions and interrupts, relevant behavior of programs often only emerges over multiple iterations of a single function.

Other Solutions Supporting Developers' Mental Model. Other tools for embedded programming include flow-based solutions [19], like FlowBoard [5], which is primarily designed for educational use. FlowBoard lets developers code on an iPad connected to breadboards. Code is "written" using connected nodes and executed live on changes. The authors focused on making coding and building hardware as *seamless* as possible. A *seamless* system has high immediacy between its hardware and software parts and therefore reduces the gap between them. The editor shows probes both on individual input-pins and nodes, to help developers understand the live runtime.

Inspection and Control via Simulation. As described in subsection 4.5, examples in MµSE necessarily combine state in software, over which we have full control, and state in the physical world, over which we have little to no control. A common solution to this problem is to use simulations, for example via Gazebo [10]. Specifically for our use cases, where unreliability of hardware is one of the most common issues, simulations may only assist in an initial design phase, until the quirks of the concrete hardware platform have to be considered.

Programming Tools Supporting Exploration and Prototyping. The activities we observed developers perform in both our studies are comparable to those of the opportunistic programmer [4]. Developers consider some code impermanent and choose trade-offs between long-term maintainability and fast iteration to gain insights. Similarly, a variety of tools exist to support exploratory programming. For example, Juxtapose [7], CoExist [25], or Variolite [9] support developers in finding optimal values during prototyping or exploration of domains by allowing the fast exploration of alternatives.

9 CONCLUSION

This paper presented MµSE, an example-based live-programming environment for prototyping embedded programs. MµSE supports the developer's existing workflow, as described by participants of our formative study, of aligning their mental model with actual execution through experimentation with code. Rather than resorting to commenting code and print statements to enable isolated experimentation within the context of their implementation, MµSE offers first-class examples and automatic probes. In an exploratory user study, we found that the participants' workflow was well-supported by MµSE, and especially automatic probes were praised. We also noticed some areas where more work is needed, for instance for inspecting and managing the physical state of the hardware. Through two expert interviews, we found that MµSE's concepts would be appreciated in a professional setting but in particular impact on performance and distraction through always-on visualizations should be investigated. In conclusion, we believe MµSE shows how a workflow for embedded software could be realized that supports embedded programmers to perform faster, more effective prototyping.

REFERENCES

[1] Amichi Amar. 2010. *Support for Resource Constrained Microcontroller Programming by a Broad Developer Community.* Ph. D. Dissertation. USA. Advisor(s) Krintz, Chandra. AAI3439403.

[2] Kent Beck. 2002. *Test Driven Development: By Example.* Addison-Weslay Longman Publishing Co., Inc., Bostan, MA, USA.

[3] Tracey Booth, Simone Stumpf, Jon Bird, and Sara Jones. 2016. Crossed Wires: Investigating the Problems of End-User Developers in a Physical Computing Task. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16).* Association for Computing Machinery, New York, NY, USA, 3485–3497. https://doi.org/10.1145/2858036.2858533

[4] Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. 2008. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. In *Proceedings of the 4th International Workshop on End-User Software Engineering* (Leipzig, Germany) *(WEUSE '08).* Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1370847.1370848

[5] Anke Brocker, René Schäfer, Christian Remy, Simon Voelker, and Jan Borchers. 2023. Flowboard: How Seamless, Live, Flow-Based Programming Impacts Learning to Code for Embedded Electronics. *ACM Trans. Comput.-Hum. Interact.* 30, 1, Article 2 (mar 2023), 36 pages. https://doi.org/10.1145/3533015

[6] Jonathan Edwards. 2004. Example Centric Programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 124. https://doi.org/10.1145/1028664.1028713

[7] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology* (Monterey, CA, USA) *(UIST '08)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/1449715.1449732

[8] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (<conf-loc>, <city>Montreal QC</city>, <country>Canada</country>, </conf-loc>) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174106

[9] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[10] N. Koenig and A. Howard. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, Vol. 3. 2149–2154 vol.3. https://doi.org/10.1109/IROS.2004.1389727

[11] Anis Koubaa. 2016. *Robot Operating System (ROS): The Complete Reference (Volume 1)* (1st ed.). Springer Publishing Company, Incorporated.

[12] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 5–8. https://doi.org/10.1109/VLHCC.2014.6883013

[13] Chao Li, Rui Chen, Boxiang Wang, Zhixuan Wang, Tingting Yu, Yunsong Jiang, Bin Gu, and Mengfei Yang. 2023. An Empirical Study on Concurrency Bugs in Interrupt-Driven Embedded Software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1345–1356. https://doi.org/10.1145/3597926.3598140

[14] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

[15] Henry Lieberman and Christopher Fry. 1995. Bridging the Gulf between Code and Behavior in Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '95)*. ACM Press/Addison-Wesley Publishing Co., USA, 480–486. https://doi.org/10.1145/223904.223969

[16] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *The Art, Science, and Engineering of Programming* 1, 2 (April 2017). https://doi.org/10.22152/programming-journal.org/2017/1/13

[17] Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/2509578.2509585

[18] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and Checking Behavior of Embedded Systems across Hardware and Software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/3126594.3126658

[19] Christopher Métrailler and Pierre-André Mudry. 2015. ESPeciaL: An Embedded Systems Programming Language. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala* (Portland, OR, USA) *(SCALA 2015)*. Association for Computing Machinery, New York, NY, USA, 51–55. https://doi.org/10.1145/2774975.2774982

[20] Michael J. Pont. 2001. *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*. ACM Press/Addison-Wesley Publishing Co., USA.

[21] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (feb 2019). https://doi.org/10.22152/programming-journal.org/2019/3/9

[22] Patrick Rein, Tom Beckmann, Eva Krebs, Toni Mattis, and Robert Hirschfeld. 2023. Too Simple? Notions of Task Complexity used in Maintenance-based Studies of Programming Tools. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE. https://doi.org/10.1109/icpc58990.2023.00040

[23] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (jul 2018). https://doi.org/10.22152/programming-journal.org/2019/3/1

[24] Hans-Christoph Steiner. 2018. Firmata : Towards Making Microcontrollers Act Like Extensions of the Computer. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. Zenodo, 125–130. https://doi.org/10.5281/zenodo.1177689

[25] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. 2012. CoExist: Overcoming Aversion to Change. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/2384577.2384591

[26] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE. https://doi.org/10.1109/live.2013.6617346

[27] J. Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (01 1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153 arXiv:https://academic.oup.com/comjnl/article-pdf/34/2/153/1400604/340153.pdf

[28] Edward R. Tufte. 1986. *The visual display of quantitative information*. Graphics Press, USA.

[29] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (apr 1997), 38–43. https://doi.org/10.1145/248448.248457

[30] Jeremy Warner, Ben Lafreniere, George Fitzmaurice, and Tovi Grossman. 2018. ElectroTutor: Test-Driven Physical Computing Tutorials. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/3242587.3242591