Latch: Enabling Large-scale Automated Testing on Constrained Systems

Tom Lauwaerts[a,*], Stefan Marr[b], Christophe Scholliers[a]

[a]*Department of Applied Mathematics, Computer Science and Statistics, Universiteit Gent, Krijgslaan 281, S9, 9000 Ghent, Belgium*
[b]*School of Computing, University of Kent, CT2 7FS Kent, United Kingdom*

Abstract

Testing is an essential part of the software development cycle. Unfortunately, testing on constrained devices is currently very challenging. First, the limited memory of constrained devices severely restricts the size of test suites. Second, the limited processing power causes test suites to execute slowly, preventing a fast feedback loop. Third, when the constrained device becomes unresponsive, it is impossible to distinguish between the test failing or taking very long, forcing the developer to work with timeouts. Unfortunately, timeouts can cause tests to be flaky, i.e., have unpredictable outcomes independent of code changes. Given these problems, most IoT developers rely on laborious manual testing.

In this paper, we propose the novel testing framework *Latch* (Large-scale Automated Testing on Constrained Hardware) to overcome the three main challenges of running large test suites on constrained hardware, as well as automate manual testing scenarios through a novel testing methodology based on debugger-like operations—we call this new testing approach *managed testing*.

The core idea of *Latch* is to enable testing on constrained devices without those devices maintaining the whole test suite in memory. Therefore, programmers script and run tests on a workstation which then step-wise instructs the constrained device to execute each test, thereby overcoming the memory constraints. Our testing framework further allows developers to mark tests as depending on other tests. This way, *Latch* can skip tests that depend on previously failing tests resulting in a faster feedback loop. Finally, *Latch* addresses the issue of timeouts and flaky tests by including an analysis mode that provides feedback on timeouts and the flakiness of tests.

To illustrate the expressiveness of *Latch*, we present testing scenarios representing unit testing, integration testing, and end-to-end testing. We evaluate the performance of *Latch* by testing a virtual machine against the WebAssembly specification, with a large test suite consisting of 10,213 tests running on an ESP32 microcontroller. Our experience shows that the testing framework is expressive, reliable and reasonably fast, making it suitable to run large test suites on constrained devices. Furthermore, the debugger-like operations enable to closely mimic manual testing.

Keywords: Automated Testing, Embedded Devices, Flaky Tests

1. Introduction

Software testing for constrained devices, still lags behind standard best practices in testing. Widespread techniques such as automated regression testing and continuous integration are much less commonly adopted in projects that involve constrained hardware. This is mainly due to the heavy reliance on physical testing by Internet of Things (IoT) developers. A 2021 survey on IoT development found that 95% of the developers rely on manual (physical) testing [41]. Testing on the physical hardware poses three major challenges, which hinder automation and the adoption of modern testing techniques. First, the *memory constraints* imposed by the small memory capacity of these devices makes it difficult to run large test suites. Second, the

*Corresponding author

processing constraints of the hardware causes tests to execute slowly, preventing developers from receiving timely feedback. Third, *timeouts and flaky tests* pose a final challenge. When executing tests on constrained hardware it is not possible to know when a test has failed or is simply taking too long.

To circumvent the limitations of constrained hardware, simulators are sometimes used for testing IoT systems [8]. Their usage makes adopting automated testing and other common testing practices much easier. Unfortunately, simulators can never fully capture all aspects of real hardware [61, 32, 13]. Therefore, to fully test their applications, IoT developers have no other option than to test on the real devices. This is the primary reason why developers still prefer physical testing. Another reason is the lack of expressiveness when specifying tests in automated testing frameworks. Testing frameworks with simulators almost exclusively focus on unit testing, and hence provide no good alternative to end-to-end physical testing performed by developers manually [67].

In this paper, we argue that programmers should not be limited by either the constraints of the hardware, or a simulator imposed by the testing framework. Therefore, our goal is to design and implement a testing framework for automatically running large-scale versatile tests on constrained systems. This has lead to the development of the *Latch* testing framework (Large-scale Automated Testing on Constrained Hardware). *Latch* enables programmers to script and run tests on a workstation, which are executed on the constrained device. This is made possible by a novel testing approach, we call *managed testing*. In this unique testing approach, the test suite is split into small sequential steps, which are executed by a testee device under the directions of a controlling tester device. The workstation functions as the tester which maintains full control over the test suite. Only the program under test—not the entire test suite—will be sent to the constrained device, the testee. The tester will use instrumentation to manage the testee and instruct it to perform the tests step-by-step. This means the constrained testee is not required to have any knowledge of the test suite being executed. This is quite different from traditional remote testing, where the entire test suite is sent to the remote device. The instrumentation of the testee is powered by debugging-like operations, which allow for traditional whitebox unit testing, but also enables the developer to write debugging-like scripts to construct more elaborate testing scenarios that closely mimic manual testing on hardware.

The research question we seek to answer in this paper, is whether the managed testing approach, i.e. splitting tests into sequential steps, is sufficient for executing large-scale tests on microcontrollers. To answer this question, we will show how managed testing allows *Latch* to overcome all three major challenges of testing on constrained devices. The approach can be summarized as follows. In *Latch* test suites are split up into smaller test instructions that are sent incrementally to the managed testee, thereby freeing the test suites from the *memory constraints* of the hardware. This is crucial in enabling large-scale test suites on microcontrollers, such as the large unit testing suite containing 10,213 tests we use to evaluate our approach. To overcome the *processing constraints*, *Latch* can skip tests that depend on previously failing tests resulting in a faster feedback loop. Finally, *Latch* handles *timeouts* automatically, and includes an analysis mode which reports on the *flakiness of tests*.

Contributions:

- We define a test specification language for writing large tests suite for constrained devices.
- We develop the *Latch* framework, that implements the test specification language as an embedded domain-specific language (EDSL).
- We present a novel testing methodology based on debugging methods, that allows common manual testing of code on hardware to be automated.
- We illustrate how *Latch* can be used to address testing scenarios from all three layers of the testing pyramid [10].
- We evaluate *Latch* by using it to run 10,213 unit tests on an ESP32 microcontroller.

The rest of the paper starts with a discussion of the challenges of testing on constrained device in Section 2. In Section 3, we give a first introduction to the *Latch* test specification language through a basic example, and use the example to give an overview of the *Latch* framework. We discuss the details of the

language in Section 4, and focus on how tests are written and executed by the framework. For each aspect of the test specification language we discuss how it helps *Latch* to address the challenges outlined previously. We conclude the section by briefly touching on the prototype implementation. Section 5 further illustrates how *Latch* can be used to handle different testing scenarios, and can help testers implement a range of testing methodologies. We discuss three scenarios, classic large-scale unit testing, integration testing, and automating physical end-to-end testing using the debug-like operations provided by *Latch*. In Section 6, we evaluate the runtime performance of *Latch* based on a variety of test suites, and present empirical evidence that managed testing enables large-scale automatic testing on constrained hardware. In Section 7, we discuss the related works, before concluding in Section 8.

2. Challenges of Testing on Constrained Devices

This section outlines the challenges preventing large-scale testing on constrained hardware.

2.1. Memory Constraints

In this article we focus on the ESP32 microcontroller family[1] having about 400 KiB SRAM and 384 KiB ROM, typically operating at a clock frequency around 160-240 MHz. Due to these hardware limitations, programs cannot be arbitrarily large as the program memory is quite small and they execute slower than workstations. For companies producing IoT devices it is often desirable to make use of the cheapest and most minimal hardware possible that can handle the task at hand. This means that when executing on the hardware, there are often very few resources to spare, which limits the ability to test the applications on the device.

When test suites become large, executing these test suites on the hardware is often not possible because the compiled binary is too big to fit in the program memory of the microcontroller. The only option then is to split the test suite into smaller parts which can fit on the device. Current testing frameworks, however, do not provide automated support for splitting large test suites and executing them incrementally on the hardware. Programmers who want to execute large test suites thus have to manually partition the test suite, execute the test on the hardware, read out the results and process the dump of the individual parts.

Finally, even when the testing framework supports partitioning of the test suite reflashing the hardware for every partition is quite time-consuming. To change the program executing on the hardware the programmer needs to flash the microcontroller, i.e. write the program in the ROM partition of the microcontroller. Depending on the microcontroller, synchronization and flashing of a new program can take several seconds making it undesirable to flash the microcontroller often.

2.2. Processing Constraints

When relying on regression testing, the programmer wants a tight feedback loop. Ideally, the entire test suite is run after each change, but this requires feedback to be reported quickly. However, by testing on constrained devices, executing the test suite can take a lot of time, slowing down the software development cycle significantly. To provide feedback as early as possible, the framework should catch failures early. This can take many forms, but in essence a failure of any kind during a test should be visible to the developer as soon as possible. Additionally, to avoid spending time on tests that cannot succeed, the framework should run as few of these tests as possible.

Finally, when multiple hardware testbeds are available it should be easy for the developer to run tests in parallel to speed up testing. The same facilities for scheduling and parallelization options available for unconstrained devices, should be integrated into testing frameworks for constrained devices.

[1] ESP32 devices can have different amounts of memory, but the order of magnitude is the same.

2.3. Timeouts and Flaky Tests

Due to the limited memory and processing power of constrained devices, large test suites need to be split up in smaller chunks. Moreover, the results of the test need to be communicated with a test machine and combined. Unfortunately, this approach implies that test engineers suddenly need to take into account many of the problems associated with distributed computing.

First, when the test machine is waiting for a response, it cannot reliably distinguish between a failure or a delayed response. Many other testing frameworks need to deal with this problem, especially JavaScript frameworks [17] where asynchronous code is prevalent [14]. These frameworks time out tests that take too long, unfortunately, the fact that a test timed out does not provide much information for developers, especially when a test includes multiple asynchronous steps.

Second, the non-determinism of the asynchronous communication also contributes to an inherent problem of testing, flaky tests [36]. These are tests that can pass or fail for the same version of the code. Unfortunately, on constrained hardware, many tests have the potential to become flaky due to the inherent non-determinism of these systems. For example, when testing communication with a remote server small changes in the communication timing with the server could lead to different behavior.

3. Managed Testing with *Latch* by Example

To overcome the outlined challenges, *Latch* uses a unique testing approach that consists of a declarative test specification language to describe tests, and a novel test framework architecture to run tests. We refer to our new approach as *managed testing*. In managed testing, the testing framework runs on a local machine and delegates tests step-by-step to one or more external platforms, which are running the software under test. To facilitate this approach, tests must be easily divisible into sequential steps. That is why *managed testing* specifies tests in a declarative test specification language, where tests are described as scenarios of incremental steps. In this section we give a first overview of how managed testing in *Latch* works through an example, before going into further detail in Section 4. The example is chosen as a small primer on how programmers can write traditional unit tests with *Latch*'s test specification language.

3.1. The Example

We define a unit test that verifies the correctness of a function for 32-bit floating point multiplication, shown in Listing 1. All example programs are written in AssemblyScript [65], one of the languages supported by *Latch*'s current microcontroller platform.

```
1 export function mul(x: f32, y: f32): f32 {
2     return x * y;
3 }
```

Listing 1: A `mul` function that multiplies its two arguments, written in AssemblyScript.

Listing 2 shows a simple test in *Latch* containing one unit test for the target program in Listing 1. *Latch*'s declarative test specification language is implemented as an embedded domain specific language (EDSL) in TypeScript [44]. Test scenarios are presented in *Latch* as TypeScript objects that have a title, the path to the program under test, and a list of steps. These steps make up the test scenario, and will be performed sequentially. Each step performs a single instruction, and can perform several checks over the result of that instruction.

The example performs only a single instruction, it requests that the *mul* function is invoked with the arguments 6 and 7 (see Line 6). These arguments are first passed to the *WASM.f32* function, to indicate the expected type in *AssemblyScript*. On Line 7, the example specifies that the function returns the number 42. Usually, the instruction and expectations for a step are described as objects, but *Latch* provides a handful of functions to construct these objects for common patterns—such as *invoke* and *returns*. This makes test scenarios less verbose, and quicker to write. We go into further detail on the structure of the *instruction* and *expectation* objects in Section 4.

```
const multiplicationTest: Test = {
    title: "example test",
    program: "multiplication.ts",
    steps: [{
        title: "mul(6,7) = 42",
        instruction: invoke("mul", [WASM.f32(6), WASM.f32(7)]),
        expect: returns(WASM.f32(42))
    }]
};
```

Listing 2: A *Latch* scenario defining a unit test for the `mul` function.

Similar to other testing frameworks, *Latch* allows test scenarios to be grouped into test suites. Crucially, the test suites in *Latch* have their own set of testee devices, on which they will be executed. When writing a new test suite in *Latch*, programmers need to add at least one testee to the suite. Such testees can range over a wide variety of microcontrollers, as well as local simulator processes. Each platform may differ in how software is flashed, or communication initialized and performed. These platform specific concerns are captured by a single TypeScript class, `Testee`. Each connection with a constrained device is represented by an object of such a class. In Listing 3 for instance, we use the Arduino platform to connect to an ESP32 over a USB port, as shown on Line 2. This line also sets the default timeout for the testee to 5 seconds. Users can add their own platforms by defining new subclasses of the `Testee` class, which can handle the specific communication requirements of the new platform.

Aside from testees, a test suite also requires test scenarios to execute. The example multiplication test is added to the test suite on Line 4, before the suite is given to *Latch* to be run on Line 5.

```
const suite = latch.suite("Example test suite");
suite.testee("wrover A", new ArduinoSpec("/dev/ttyUSB0", "esp32:esp32:esp32wrover"), 5000)
     .testee("wrover B", new ArduinoSpec("/dev/ttyUSB1", "esp32:esp32:esp32wrover"))
     .test(multiplicationTest);
latch.run([suite]);
```

Listing 3: *Latch* setup code to run the `multiplicationTest` on two ESP32 devices.

Listing 3 shows how a test suite is built in *Latch* through a fluent interface [69], meaning the methods for constructing a test suite can be chained together. Each test suite in latch is entirely separate from the rest, and therefore contains only its own tests, and platforms to run those tests on. In the example, two ESP32 devices are configured for the test suite. This means that when the test suite is started with the *run* function on Line 5, the framework will execute all scenarios in the suite on all configured platforms. Alternatively, the user can configure *Latch* to not execute duplicate runs, but instead to split the tests into chunks that are performed in parallel on different devices. In that case, each test is only run once and the execution time of the whole test suite should improve due to the parallelization.

3.2. Running the Example on the Latch Architecture

To run the above testing scenario on a remote constrained device, the test is loaded into *Latch* on the local unconstrained device, the **tester**. During testing, the **tester** manages one ore more **testees** (constrained devices) to execute tests step-by-step. Figure 1 gives an overview of all steps and components involved during testing in the *Latch* framework. The left-hand side shows the tester, which runs the *Latch* **interpreter** and **test execution platform**. The interpreter component is responsible for interpreting the test suites, which are written in the **test specification language**, while the test execution platform sends each instruction in a test step-by-step to the testee device over the available communication medium. The test execution platform also parses the result, and handles all other aspects of communication with the testee device.

We will go over the steps shown in Fig. 1 in the order they are executed by *Latch*. Running a test suite is initiated by the interpreter, which takes the test suite specification ①, and schedules the **scenarios** ②. Since the example test suite in Listing 3 only contains a single test scenario, the multiplication test, with a single step—the scheduling is not relevant in this case. In real test suites, the order in which tests are run is important, it can help detect failing tests early, or minimize expensive setup steps. When the interpreter

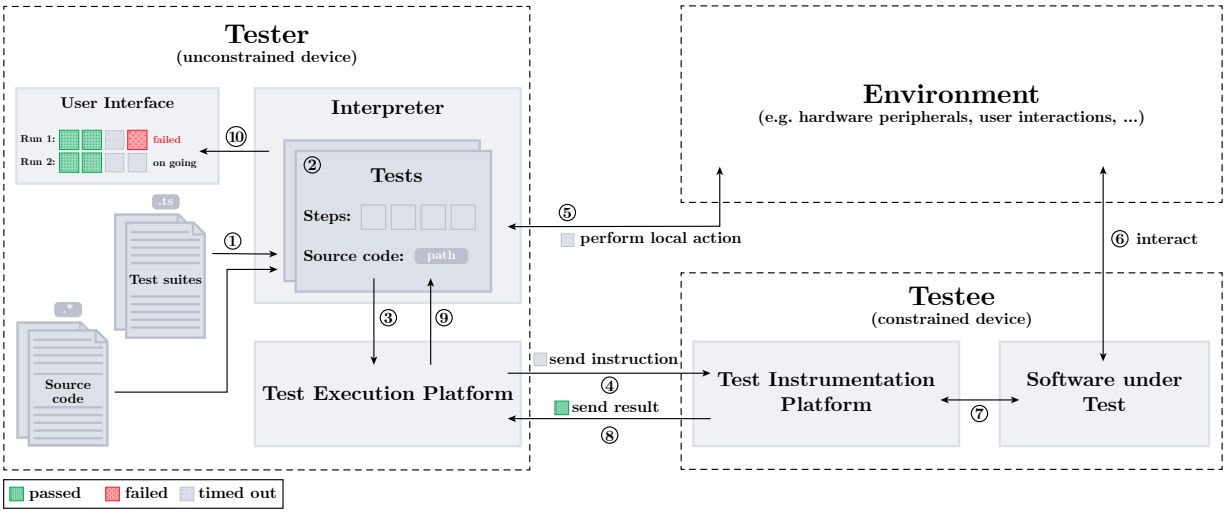

Figure 1: Schematic overview of the interaction between components in *Latch* during a test.

selects a test to be executed, it will instruct the test execution platform ③ to first upload the **software under test**, and subsequently sends the instructions of the scenario to the **test instrumentation platform** ④. In the case of our example, *Latch* compiles the *multiplication.ts* file and uploads it to the ESP32 device that is connected to the USB port. Once this step is completed, *Latch* sends the invoke instruction to the testee, which will execute the *mul* function with the supplied arguments.

Aside from forwarding instructions to the test instrumentation platform, the tester can also perform custom actions to control the **environment** ⑤. For instance, these actions can control hardware peripherals, such as sensors and buttons, that interact with the constrained testee ⑥ during the test.

Listing 4 shows how a step might send an MQTT message to a server as an example of an action that acts on the environment. Such a step, could be useful when testing an IoT application that relies on MQTT messages. The microcontroller can connect to an actual testing server, and via custom actions *Latch* can test if the device responds correctly.

```
1 const sendMQTT: Step = {
2     title: "Send MQTT message",
3     instruction: simpleAction((): void => {
4         let client: mqtt.MqttClient = mqtt.connect("mqtt://test.mosquitto.org");
5         client.publish("parrot", "This is an ex-parrot!");
6     })
7 };
```

Listing 4: An example *Latch* step, which performs a custom action that sends an MQTT message to a server.

In contrast with Listing 2, this example constructs the instruction object explicitly, rather than calling a function such as *invoke*. There are two types of instructions, they can be either a *request* to the test instrumentation platform, such as the invoking of a function, or a custom *action*. In this example we construct a simple action that takes no arguments and returns nothing. Actions allow tests to execute TypeScript functions as steps in the test scenario, in this case the function simply publishes a test message to the MQTT server (Line 5). We go into further detail on the types of actions and requests in Section 4.

As tests are performed, the software under test is controlled by the test instrumentation platform in accordance with the *request* instructions send by the test execution platform ⑦. In other words, the test instrumentation platform will receive the command from the tester to execute the *mul* function, and make the software under test invoke it. The instrumentation of the software under tests, allows the test instrumentation platform to return any generated output to the test execution platform ⑧. Whenever the tester

sends an instruction to the testee, *Latch* will wait until the testee returns a result for the instruction. When working with constrained devices, communication channels may be slow or fragment messages. *Latch* takes care of these aspects automatically.

As part of a step, the scenario description can specify a number of assertions over the returned results. In the example, we require that the *mul* function returns 42, as specified on Line 7 of Listing 2. Once the expected output is received by the tester, *Latch* checks all assertions against it. These assertions are verified by the interpreter ⑨, before the result of the step is shown in the **user interface** as either passed, failed, or timed out ⑩. For example, after the test instrumentation platform returns the result of the *mul* function, *Latch* will check if it indeed equals 42 and report the result.

A step can have three kinds of results; either it timed out, or all its assertions passed, or one or more assertions failed. In other words, step is marked as failing when at least one assertion fails. If no assertions were included in the step, *Latch* will not wait for output, and immediately report the action as passing. When the testee fails to return a result after a preconfigured period, it is marked as timed out. Similarly, a scenario is marked as failing when at least one step fails. When a step fails, the test execution platform will—by default—continue the scenario without retrying the step. This is useful when the steps in the scenario are independent of each other to gather more complete feedback. Otherwise, developers can configure *Latch* to abort a scenario after the first failure.

The results of each step are reported while the test suite is executing. When the entire suite has run, *Latch* will give an overview of all the results for both the steps and the test scenarios. This overview includes, the number of passing/failing tests, the number of passing/failing steps, the number of steps that timed out, and the overall time it took to run the suite. In addition, the developer can configure *Latch* to report on the flakiness of the test by executing the tests multiple times. This way, *Latch* can compare the results of different runs to give developers more insight into the flakiness of their test suites. As Fig. 1 shows under the user interface component, the results in this case will be reported for each run separately. Whenever the runs give different results, the scenario is marked as flaky and the failure rate is reported.

3.3. From Small Examples Towards Large-scale Test

The running example in this section illustrates *Latch*'s basic testing features. In particular, how *Latch* divides tests into small steps that are executed sequentially. This means that the size of the test suite is no longer constrained by the memory size of the embedded device. While the example here only includes a single step, one can easily imagine test cases that require many more steps. Let us suppose we stay within the realm of unit testing a mathematical framework. We can imagine a more complicated mathematical operation than multiplication that requires thorough testing, for instance a function *eig* for calculating the eigenvalues of a matrix. In this case the test scenario would include many steps, that each invokes the *eig* function with a different matrix. This is similar to the large-scale unit testing suite we will discuss in Section 5, and those run as part of the evaluation in Section 6.

Section 5 discusses realistic examples for each layer of the testing pyramid; unit testing, integration testing, and end-to-end testing. The examples will illustrate how using small steps powered by debugging-like operations, uniquely enables *Latch* to test remote debuggers and automate IoT scenarios and manual hardware tests. For example, it becomes much easier to test whether a microcontroller successfully receives asynchronous messages from a remote server, and handles these message correctly. The test can set breakpoints in the code that is expected to be executed when a message arrives. Before sending the message, the test can pause the execution at the exact place in the program, it wants the message to be received. The *Latch* instructions allows users to write these kinds of testing scenarios in a convenient way. Moreover, the increased control over the program, makes the test scenarios much easier to repeat reliably under the same conditions.

4. The *Latch* Test Specification Language

Latch tests are written in a declarative test specification language embedded in TypeScript. This EDSL allows developers to specify what tests should be performed, while hiding the complexity of communicating

with the constrained testing device. Equally important are the debug-like commands provided by the language, which make it easier to automate hardware testing scenarios. Latch tests can be viewed as scripted scenarios of sequential operations. The programmer can specify what the result of executing an operation should look like, instead of manually testing whether the returned value is consistent with the expected result. For more complex tests the programmer can write test-specific evaluation functions to check whether the program behaves as expected.

The test specification language consists of four major abstractions: a test, a testing step, test instructions, and assertions. Each test includes a name, some start-up configuration and the testing steps which need to be executed during the actual test. Each testing step specifies an instruction that needs to be executed. There are two types of instructions, commands and actions. The commands are debug-like operations that are send directly to the test instrumentation platform of the testee, such as invoking a method, pausing the program, etc. Alternatively, there is support for user-specified instructions called actions. These actions allow programmers to implement their own logical and physical interactions with the hardware or the environment.

The interface of a test, shown in Listing 5, consists of a title, the path to the program to load on the testee device, a set of initial breakpoints to halt execution, a list of dependent test, and a set of steps to be executed during the test. Both the initial breakpoints and dependent tests are optional, as indicated by the question mark after their identifier.

```
interface Test {
  title: string;
  program: string;
  steps: Step[];
  dependencies?: Test[];
  initialBreakpoints?: Breakpoint[];
}
```

Listing 5: Interface for *Latch* tests. Each test has a title, indicates a program to be tested, and lists the steps to executed.

Testing steps all adhere to the *Step* interface shown in Listing 6. Each step should minimally have a title and specify which instruction to perform when executed. A step only contains a single instruction, and all steps are executed synchronously. As part of a step, the result of executing an instruction can be verified by means of assertions.

```
interface Step {
    readonly title: string;
    readonly instruction: Command<any> | Action<any>;
    readonly expect?: Assertion[];
}
```

Listing 6: A step has a name, a specific command or action it should perform, and a possibly list of assertions to check.

An instruction in *Latch* is either a command, or an action. Both instruction types are annotated with their return type, this is the type of the object passed to each assertion of the step. The list of assertions is optional, a step without any assertions will always succeed and immediately go to the next step.

4.1. Default Commands in Latch

The set of commands *Latch* supports is shown in Table 1. We divide the set of commands in intercession, meta, and introspection commands. The intercession commands, allow *Latch* tests to intervene directly with the software under test. With *invoke* the programmer can call a function and wait for the result, as illustrated by the step in our multiplication example (Listing 2). This enables unit testing of specific functions, as is the popular approach adopted in most testing frameworks [58, 66]. With *set local* the programmer can change a local variable, this is especially useful to test a program with local boundary conditions without having to rerun the program completely.

The *reset* and *upload module* instructions are primarily for internal use in *Latch*, but are available in the test specification language. The upload module instruction loads a binary onto the testee, replacing any current program. The reset instruction restarts the current program.

Category	Commands
Intercession	invoke, set local, *upload module*
Meta	pause, set breakpoint, continue, delete breakpoint, step, step over, *reset*
Introspection	core dump, dump callback mapping, dump locals

Table 1: The *Latch* commands. Internal commands are in italic.

The meta instructions allow the programmer to install a debugging scenario by setting breakpoints and running the program to a particular point in the execution. These are especially useful for automating manual hardware tests, where different steps and events often need to happen in very specific orders. By controlling the execution of the program, these kinds of scenarios can be replicated accurately each time.

Finally, the introspection commands allow the programmer to inspect the current state of the program. Without these commands, *Latch* test would be limited to testing black boxes, since the software under test is executed on a different device. Thanks to the introspection commands, *Latch* supports black box as well as white box tests.

The proposed set of commands are inspired by standard debugging instructions, and focus on enabling standard unit testing, as well as automation of manual hardware tests. Since the test specification language is embedded in TypeScript, the set of commands is easily extended by the user. Other debugging instructions could similarly inspire new *Latch* commands, such as run until, setting of conditional breakpoints, exception breakpoints, or inspecting memory addresses. Instructions tailored to asynchronous tests, such as awaiting an event, or waiting for a given time, would likewise be good additions. A new command has to implement the interface shown in Listing 7. A command is identified by the test instrumentation platform by its type, examples include pause, set breakpoint, and step. These commands can optionally take a payload, such as a breakpoint address for example, and each command has its own parser to interpret the response of the test instrumentation platform.

```
1 export interface Command<R> {
2     type: Interrupt,                    // type of the debug message (pause, run, step, ...)
3     payload?: (map: SourceMap.Mapping) => string,  // optional payload of the debug message
4     parser: (input: string) => R                              // the parser for the response
5 }
```

Listing 7: Commands are distinguished by `type` and may have callback to access payload. Results are extracted by a parser.

By taking inspiration from debugging instructions, managed testing permits for a wide range of automated tests to be implemented, which would otherwise require additional engineering efforts in existing unit testing frameworks. Additionally, we have found that it provides a very natural way of writing tests for constrained devices. We illustrate both these points by discussing in-depth examples for each layer of the testing pyramid in Section 5.

4.2. Custom Actions in Latch

Aside from these commands, *Latch* allows steps to perform custom actions. These custom actions enable developers to execute arbitrary code as part of a step in the testing scenario. This is useful for interacting with the environment when testing the firmware of hardware components. Listing 8 shows the interface for a custom action. An action is an object with a single act field, containing a function that takes a Testee argument and returns a promise. The testee argument is provided at runtime by the *Latch* framework, to provide custom actions with access to the test instrumentation platform. This is useful to define actions that need to respond to changes on the testee device, for instance waiting for a breakpoint to be hit. Actions may be asynchronous and therefore return promises. A promise is the standard mechanism for managing asynchronicity in JavaScript and TypeScript [52, 40]. If the action is expected to return a response, the promise should contain the output. For *Latch* to run checks over this output, it needs to be of the *Assertable* type. As shown on Line 1 in Listing 8, an *Assertable* is an object that contains any number of properties that are indexed by strings. *Latch* provides a function that can turn any object into an *Assertable* object.

```
type Assertable<T extends Object | void> = {[index: string]: any};

interface Action<T extends Object | void> {
    act: (testee: Testee) => Promise<Assertable<T>>;
}

declare function assertable<T extends Object>(obj: T): Assertable<T>;
```

Listing 8: *Latch* actions allow developers to execute arbitrary code in a test step. Output of such actions can be checked for correctness with the `Assertable<T>` interface.

In Section 3, we briefly showed a simple action in Listing 4. However, this action returned no result, over which the test step could define assertions. Listing 9 gives an second example of an action that does return a result. The action will listen for the next MQTT message for a specific topic. On Line 5, the `act` function returns a promise that resolves when the first message for the correct topic arrives. The promise contains the MQTT message of the application-specific *Message* type, including a topic and payload field. This object can be used to define checks over the payload of the message with *Latch* assertions. However, for *Latch* to run checks against the message, the returned object must conform to the *Assertable* interface. That is why on Line 8, the message object is wrapped in a *Assertable* by the assertable function, shown in Listing 8.

```
function listen(topic: string): Action<Message> {
    let client: mqtt.MqttClient = mqtt.connect("mqtt://test.mosquitto.org");

    return {
        act: () => new Promise<Assertable<Message>>((resolve) =>
            client.on("message", (_topic: string, payload: Buffer) => {
                if (topic === _topic)
                    resolve(assertable({topic: topic, payload: payload.toString()}));
})) }; }
```

Listing 9: An example of a pure action that listens for the next MQTT message to a specific topic.

4.3. Assertions over instruction results

Aside from the instruction, each step contains a list of zero or more assertions. These assertions are used to perform checks on the result of the step's instruction. The result of an instruction is always of the *Assertable* type shown in Listing 8.

For each string-indexed property of an *Assertable* result, a test step can contain one or more assertions. The interface of the assertions is shown in Listing 10. The Assertions represent a check over a single property of the assertable object, specified by their string index. The assertions over the object's properties follow the `Expect` interface, also shown in Listing 10. An `Expect` object represents an assertion over an object property of the result, and takes a type parameter `T` that should correspond with the type of that property. The `Expect` interface can be used to check for a value of type `T`, or a behavior encoded by the `Behavior` enum also shown in Listing 10. Behaviors can check for an unchanging, changing, increasing, or decreasing value. If these options do not suffice, developers can write their own custom checks. These are written as comparison functions that take the actual resulting value from the test, and return a boolean indicating whether the check passes.

```
interface Assertion { [index: string]: Expect<any>; }

type Expect<T> = T | Behaviour | (value: T) => boolean;

enum Behavior { unchanged, changed, increased, decreased }
```

Listing 10: Instructions return their results as Assertable objects. In Latch tests specify assertions over the arbitrary properties of these Assertable result.

The interface for assertions is implemented in TypeScript using a discrimination union, which is a design pattern used to differentiate between union members based on a property that the members hold. For brevity, we have omitted this detail in Listing 10 and all examples that follow.

The introspection commands are particularly interesting for assertions, since they enable assertions over the internal state of the testee. Consider the core dump command which returns a state object, shown in Listing 11 shows the dump command, which returns a state object.

```
1 const dump: Command<State>;
2
3 interface State {
4     line: number;    // current line position
5     column: number;  // current column position
6     mode: Mode;      // execution mode
7     func: string;    // current function
8 }
```

Listing 11: The *core dump* command returns a state object, which contains the source location, execution mode, and name of the currently executing function.

For example, the dump command allows a step to check whether the testee is paused in a particular function. Listing 12 shows how you might write this test step. Line 4 adds two assertions to the step. The first checks whether the mode field in the state is set to pause, and the second checks if the current function has the correct name.

```
1 const step: Step = {
2     title: "CHECK: entered *echo* function",
3     instruction: Command.dump,
4     expect: [{mode: Mode.PAUSE}, {func: "echo"}]
5 }
```

Listing 12: Example step that uses the *core dump* command to check that execution paused in the `echo` function.

With the test specification language, developers can declaratively describe tests independently of the platform they should be executed on. By embedding the domain-specific language in TypeScript, we can use the type system of TypeScript to type all the constructs in the EDSL and catch mistakes in tests early.

4.4. Managed Testing

Given a test written in the *Latch* test specification language, the framework will execute it through a single tester which manages one or more constrained testees. That is, the software under test runs on a constrained device and the test suite is kept on the unconstrained tester device. The tester will instruct the constrained device to perform tests by sending instructions step-by-step. This design allows test to be run on constrained devices, while overcoming the memory constraints.

In the example of Section 3 we configured a test suite in *Latch* to run on two devices. The test specification language has two main components to specify this configuration. First, the language has an overarching concept of a test suite that groups a number of tests. Each test suite runs independently of the others, and maintains its own devices, and their communication. Listing 13 shows the public methods of the `TestSuite` class in *Latch* that can add new devices and tests to a test suite. Finally, when a test suite is created and fully configured, it can be executed on all devices with the `run` method.

```
1 class TestSuite {
2     public testee(name: string, testee: Testee): TestSuite;
3     public scheduler(scheduler: Scheduler): TestSuite;
4     public test(test: Test): TestSuite;
5 }
```

Listing 13: The `TestSuite` allows developers to specify the testees, i.e., target devices, configure the scheduler, and the set of tests to be executed.

The devices passed to a test suite, represent a single connection to a device. *Latch* supports different devices each with their own abstraction, which needs to be able to connect and disconnect, upload a program, and send instructions. The interface of these abstractions is captured by the abstract class in Listing 14.

```
abstract class Testee {
    abstract connect(): Promise<void>;
    abstract upload(program: string): Promise<void>;
    abstract sendCommand<R>(command: Command<R>): Promise<R>;
    abstract disconnect(): Promise<void>;
}
```

Listing 14: The `Testee` implements support for different devices to enable upload of programs, and command execution.

4.5. Using Test Scheduling and Expressing Dependent Tests

Performing tests on remote hardware testbeds is often slow, which delays feedback. To make testing on constrained devices part of continuous integration in practice, we reduce the time it takes to get feedback on failing tests by not running unnecessary ones. *Latch* allows dependencies between tests to be defined explicitly, as part of the test syntax as shown in Listing 5. Each test in *Latch* can specify a list of tests it depends on. The framework treats these dependencies between tests as transitive. This enables the framework to skip tests that cannot succeed, thereby mitigating the effects of the processing constraints.

4.5.1. Example

To illustrate test dependencies, we expand on our earlier multiplication test example. Suppose our constrained device is connected to a temperature sensor that uses Fahrenheit, but our software uses Celsius. For the conversion, we use the AssemblyScript function in Listing 15.

```
function celsius(fahrenheit: f32): f32 {
    return (fahrenheit - 32) * 0.556;
}
```

Listing 15: AssemblyScript function to convert Fahrenheit to Celsius.

The conversion to Celsius depends on the multiplication of 32-bit floating point numbers, which we tested in our previous example. If the test for multiplication fails, we know that the `celsius` function will fail, too, and we can avoid running the temperature conversion test to safe time. Consequently, we list the `multiplicationTest` as a dependency on Line 4 in Listing 16. Dependencies are entirely defined by the user, the only restriction is the disallowing of cyclical dependencies. Currently, the framework throws a runtime error whenever it encounters a cyclical dependency between a group of tests.

For complex scenarios, we can list an arbitrary number of *dependencies*. If any of the dependencies should fail, *Latch* skips the test. For continuous integration these tests are considered failing, but they are marked with a distinct *skipped* label and counted separately from true failures by *Latch*.

```
const dependentTest: Test = {
  title: "Example Test with a dependency.",
  program: "celsius.ts",
  dependencies: [multiplicationTest],
  steps: [{
    title: "Fahrenheit to Celsius test",
    instruction: invoke("celsius", [WASM.f32(46.4)]),
    assert: returns(WASM.f32(-8.0))
  }]
};
```

Listing 16: *Latch* test suite for the `celsius` function, with a dependent scenario.

Algorithm 1 The default scheduling algorithm in *Latch*.

Require: list of tests *suite*

1: *accumulator* ← [ ][ ]
2: *trees* ← findDependencyGraphs(*suite*)
3: **for all** *tree* ∈ *trees* **do**
4: *levels* ← groupSiblings(*tree*)
5: **for all** *index*, *level* ∈ *levels* **do**
6: **append** *acc*[*index*] **with** *level*
7: **end for**
8: **end for**
9: **return** flatten(*accumulator*)

Algorithm 2 The optimistic scheduling algorithm to minimizing program uploads.

Require: list of tests *suite*

1: *schedule* ← [ ]
2: *trees* ← findDependencyGraphs(*suite*)
3: **for all** *tree* ∈ *trees* **do**
4: **append** *schedule* **with** breadth-first(*tree*)
5: **end for**
6: **return** *schedule*

4.5.2. User-defined Schedulers

The order in which tests are executed can also influence the execution time of the test suite, especially since failing dependent tests can prevent unnecessary computations. To further speed up the execution, the test specification language allows developers to configure the scheduling algorithm the framework uses when running a test suite. The best scheduling algorithm depends on the exact test suites. Therefore, scheduling is configured at the level of a test suite as shown earlier in Listing 13. Scheduling algorithms are implemented as subclasses of the `Scheduler` class from the test specification language shown in Listing 17. The class only has one public method that takes a list of tests, and returns a new list with the tests sorted according to the scheduler's prioritization. This class allows developers to embed their own schedules in the test specification language.

```
1 class Scheduler {
2     public schedule(tests: Test[]): Test[];
3 }
```

Listing 17: `Schedulers` enable custom ordering of tests. The ordering can avoid unnecessary test execution, or allow for test prioritization.

The current implementation of the *Latch* framework, provides two predefined schedulers, the default and the optimistic scheduler. We give the pseudocode for both scheduling algorithms in Algorithm 1 and Algorithm 2 respectively. Since dependencies amongst tests are transitive and cyclical dependencies are disallowed, we can extract trees from a test suite, where linked nodes depend on each other. Both algorithms will use this fact.

The default scheduler prioritizes the dependencies between tests and works best with test suites where a large number of tests dependent on a much smaller set of scenarios. The algorithm of the default scheduler, starts by initializing an accumulator as a list of lists. Then, it finds all the dependency trees, as shown on Line 2. The *findDependencyGraphs* function constructs a forest of directed dependence trees. In these graphs the nodes are tests that directly depend on their parents. The function will throw a runtime error if any cyclical dependencies are encountered. After the trees are found, for each tree the tests are aggregated into lists of tests with the same depth in the tree, the siblings in other words. Subsequently, the algorithm appends each group of siblings to the list in the accumulator that corresponds to its level. After all trees have been traversed, the accumulator is flattened to a one-dimensional list, and returned as the schedule.

The optimistic scheduler is built on the assumption that dependent tests are more likely to use the same program. If this is the case for the test suite, it can result in far fewer code uploads during a run compared to the default scheduler. It constructs the dependence trees in the same way as the default scheduler. Next, the algorithm will append their tests breadth-first to the schedule. Within the same depth the tests are sorted alphabetically based on the program's name, to minimize the number of times the tester needs to upload code. The resulting list of tests, is ordered in such a way that trees are executed one after the other, and no test is ever run before any test it depends on.

The default scheduler can be seen as traversing the entire dependence forest breadth-first. In contrast, the optimistic scheduler iterates breadth-first over each dependence tree in succession. Again, at each depth the tests are sorted alphabetically according to their program. These two schedulers are provided as examples of scheduling algorithms, each test suite most likely has its own optimal algorithm.

Thanks to the scheduling based on the test dependencies, *Latch* can detect failures early and prevent unnecessary tests from running. However, the time needed for executing tests can be further minimized by executing test suites in parallel. Since microcontrollers are typically cheap and abundantly available, it makes sense to run different tests on separate devices at the same time. Currently, the schedulers still return a single ordering over the tests, but the dependency trees constructed as part of their algorithms offer an opportunity to parallelize. Different dependency trees can be safely run in parallel, since tests in different trees have no dependencies in common.

4.6. Handling and Reporting on Timeouts

Since all actions of a test are executed remotely, the tester cannot distinguish between an unresponsive test and a test that can still succeed after a long time. This is an unavoidable problem when testing in the presence of asynchronous actions. Many modern testing frameworks deal with this by adding timeouts to all asynchronous tests. In *Latch* we use timeouts for testing on constrained devices, too, but provide as much information as possible about where timeouts occur. Thus, *Latch* provides timeouts at the level of single instructions, following the example of frameworks dedicated to testing of asynchronous system [23], rather than merely at the level of a test, as is common practice in more general test frameworks [50, 58, 66]. We found that debugging timeouts, is significantly easier with fine-grained information.

Listing 18 shows an example of a test that specifies a custom timeout limit for a single step. This setting will override the default timeout limit for the testee, the test is run on.

```
{
    title: "Test with custom timeout",
    program: module,
    steps: [{
        title: "Step with custom timeout",
        instruction: invoke("sleep", [WASM.u32(1000)]),
        timeout: 1100
    }]
}
```

Listing 18: *Latch* step with custom timeout

The actions of the tests are not the only source of asynchronicity in *Latch*. There are other asynchronous actions behind the scenes, from compiling test programs to connecting with hardware testbeds. In *Latch* every asynchronous action can time out, and each timeout has their own helpful message indented to make them easily identifiable by developers.

4.7. Detecting and Reporting Flaky Tests

The asynchronicity and non-determinism introduced by *Latch* and the hardware testbeds, can cause any test to become flaky. These tests can both succeed and fail for the same version of the software under test. In *Latch*, we follow the recommendation of Harman and O'Hearn [24] to considers all tests as flaky. Indeed, flaky tests can hint at bugs. Therefore, we use an approach that improves the debuggability of flaky tests.

The framework can run in two modes. A normal mode which executes each action and each test at most once, and an analysis mode where tests are executed multiple times to analyze flakiness. In this mode we assume all tests are flaky. Therefore, tests are rerun even if they succeed. This can slow the test suite significantly, which is why it is provided as an optional mode. Indeed, the default mode still allows continuous integration to report initial results quickly, while the flakiness of the test suite can be reported at a later moment after the second mode has finished. The analysis mode trades performance for more information and certainty.

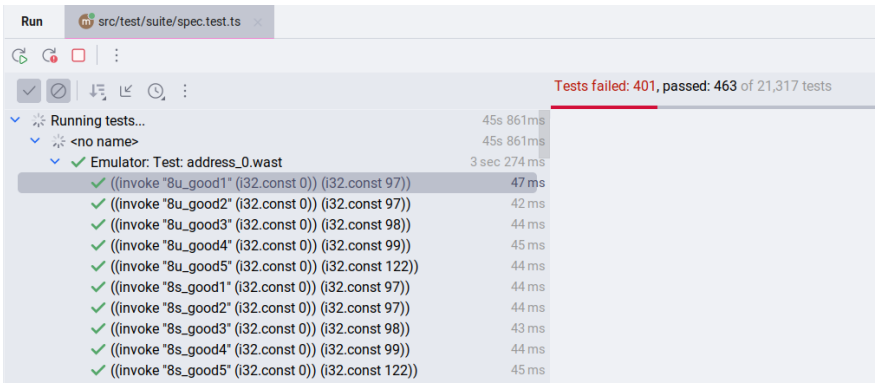

Figure 2: IDE integration in WebStorm [28].

The analysis mode can be configured with a minimum and maximum number of runs. Since we consider all tests as flaky, we will execute each test at least the minimum number of times. If the test reports the same result for each of these runs, *Latch* assumes it is not flaky and stops for this scenario. In the other case, we already have proof that the test is flaky, and *Latch* will continue executing up to the maximum number of times to get a more representative measure of the flakiness. The maximum number of runs is important to have statistically significant results, and can therefore be configured by the user for each run. The minimum and maximum number of runs can be configured by the user. When a test suite is executed on multiple platforms, flakiness is measured for each platform separately. At the end of the analysis run, *Latch* reports the global flakiness of the test suite for each platform as the number of flaky scenarios, and at the end of the analysis run, *Latch* reports the flakiness on each platform for each test and gives an overview with the overall flakiness of the test suite for each platform separately and all platforms together.

4.8. Prototype Implementation

The prototype implementation of *Latch* is a TypeScript library built on the WARDuino [21, 38] virtual machine for constrained devices and the Mocha testing framework for JavaScript and TypeScript.

WARDuino is a WebAssembly [22] virtual machine targeting ESP32 microcontrollers. The virtual machine also has basic debugging support, which we used as the basis for implementation our test instrumentation platform in *Latch*. By using a WebAssembly virtual machine, *Latch* can test programs written in any language that can compile to WebAssembly. This includes most of the mainstream programming languages used today, such as C, C++, Java, Python, Ruby, Rust [16]. In order to fully test a language, *Latch* needs to have support for compilation and sourcemapping. The current implementation has support for compiling and constructing sourcemaps for AssemblyScript.

We believe the general principles we use for implementing the *Latch* prototype on WARDuino, can be applied to any language or virtual machine which provides basic debugging support. This includes the C programming language that is supported by many microcontrollers, and which offers basic debugging by means of a JTAG interface.

The *Latch* prototype uses the Mocha testing framework for JavaScript and TypeScript to report the results of the tests in the *Latch* framework. Handling the output through an existing framework, immediately gives *Latch* integration into most of the existing IDEs used for programming in TypeScript and JavaScript.

5. Testing with *Latch*

Latch offers a framework for writing unconstrained automated test scripts, that can address many different testing scenarios. To demonstrate the versatility of *Latch*, we will present a common testing scenario for microcontrollers for each stage of the testing pyramid [10]. Several versions of the pyramid exist, often

tailored to specific software domains [46]. Generally, testing pyramids split testing into three or more stages, which are often performed in order from bottom to top. Each successive layer in the pyramid tests larger parts of the software in one test. Therefore, each layer will typically have fewer tests than those before it. The testing pyramid is a common way of representing the full scope of testing for a software project, it is therefore suitable to showcase *Latch*'s ability to support the full range of testing scenarios.

In this section, we adhere to the classic testing pyramid, with unit testing at the bottom, followed by integration (or service) testing, and finally topped by end-to-end (user) testing. We first highlight how *Latch* can perform realistic, large-scale unit testing on constrained hardware. Then we show how *Latch* can test the instrumentation it uses as an illustration of integration testing. Finally, we show how manual testing on hardware can be automated to perform end-to-end testing. The example test suite illustrates how this can be used to test both the hardware itself, and the software libraries used for controlling that hardware.

5.1. Unit Testing: Large-scale Testing of a Virtual Machine

In the testing pyramid, the largest number of tests are the unit tests. The underlying virtual machine used by *Latch*, WARDuino, uses a subset of the official WebAssembly specification test suite, to test whether it conforms with the WebAssembly standard. WARDuino does not use the entire official test suite, since it does not yet support all the latest accepted proposals to the standard. The WARDuino project uses an extended version of the virtual machine to parse and run the unit tests from the test suite. Unfortunately, this means it cannot be executed on microcontrollers, since the entire suite needs to be included as well as the large parsing library needed to extract the unit tests. By using *Latch*, we are able to take the same test suite, and execute it on an ESP32 microcontroller. We discuss the results further in Section 6. In this section, we focus on how the official specification test suite is written in *Latch*.

Test files in the WARDuino test suite contain a number of WebAssembly modules, each of which has a number of assertions. These assertions are so called *assert-return* tests, which invoke a WebAssembly function and specify the expected result. The assertions are written as S-expressions.[2] Listing 19 shows two such assertions.

```
1 (module (func (export "mul") (param $x f32) (param $y f32) (result f32) (f32.mul (local.get $x) (
      local.get $y))))
2 (assert_return (invoke "mul" (f32.const -0x0p+0) (f32.const 0x0p+0)) (f32.const -0x0p+0))
3 (assert_return (invoke "mul" (f32.const -0x1p-149) (f32.const -0x0p+0)) (f32.const 0x0p+0))
```

Listing 19: An *assert-return* test from the official WebAssembly Specification test suite, testing the `f32.mul` operation.

With *Latch*, we can run the same tests on actual embedded hardware. The structure of the WebAssembly specification test suite is well suited for *Latch*'s test specification language. The asserts coincide perfectly with the steps in the test. Each assert contains a single action to perform and a single assertion to check. Therefore, all specification tests for WebAssembly can be encoded as a single test suite with a test for each distinct module. Listing 20 shows the example in Listing 19 translated into a *Latch* test.

To test the WARDuino virtual machine, we converted the official WebAssembly test specification into a large *Latch* test suite. Since *Latch* is a DSL embedded in TypeScript, this conversion can easily be done programmatically in TypeScript code. Converting the *assert-return* S-expressions to *Latch* syntax in this way is fairly, easy. The conversion enables us to test the WARDuino virtual machine incrementally. The test instrumentation framework will only load one WebAssembly module from the test suite at a time and each test is converted into steps, which are sent to the testee incrementally, i.e. the testing steps do not need to be stored in the memory of the testee. In Section 6, we give an overview of the performance of executing this test suite on an ESP32 device.

5.2. Integration Testing: Testing a Debugger API

Due to its design, *Latch* is well suited to test the debugging operations of the WARDuino virtual machine. Testing the debugger API exemplifies the second layer of the testing pyramid: integration testing.

[2] This conforms with the official WebAssembly specification tests, which can be found on: `https://github.com/WebAssembly/spec/tree/main/test/core`

```
const test: Test = { // Spec test
  title: "Test f32.mul operation",
  program: "module.wast",
  steps: [
    { title: "assert: -0 * +0 = -0",
      instruction: Command.invoke("mul", args: [-0, 0]),
      expect: returns(WASM.f32(-0)) },
    { title: "assert: -1e-149 * -0 = 0",
      instruction: Command.invoke("mul", [-1e-149, -0]),
      expect: returns(WASM.f32(0)) }
  ]
};
```

Listing 20: The `f32.mul` test has two steps, each checking the result of `mul` on different inputs.

As an example, consider the step over debug instruction, which steps over a single function call or a single instruction when the instruction does not call a function. A simple test starts at a function call and sends the debugging instruction, before checking if the program did step over it correctly.

```
1 export function main(): void {
2     blink();
3     print("started blinking");
4 }
```

Listing 21: The blink program used by the integration test for the WARDuino debugger API.

The blink program in Listing 21 calls on Line 2 the `blink()` function and on Line 3 the `print()` function. With this program, we check that the program executes up to Line 3, rather than stopping at the start of the main function. Listing 22 shows the corresponding definition of a test in *Latch*. It loads the program, calls the main function, and sends a step over instruction. At the end of the test, it checks whether the current line has indeed moved to Line 3.

```
const stepOverTest: Test = {
    title: "Test STEP OVER",
    program: "blink.wast",
    dependencies: [dumpTest, invokeTest]
    steps: [
      { title: "Start program",
        instruction: Command.invoke("main", []) },
      { title: "Send STEP OVER command",
        instruction: Command.stepOver },
      { title: "CHECK: execution stepped over direct call",
        instruction: Command.dump,
        expect: [{line: 3}] }
    ]
};
```

Listing 22: The description for *Latch* of the *step over* test.

The debugging tests illustrate how integration tests can frequently dependent on each other. For instance, our small *step over* test uses the *invoke* and *dump* instructions, which can also be tested with *Latch*. When tests for either these two instructions fail, we can no longer rely on the results of the *step over* test. Since the *invoke* or *dump* commands may be broken, they might cause false positives, or false negatives, in tests that use them. There is no reason to run tests that cannot be trusted. In *Latch*, we can encode the dependency of the *step over* test on the *invoke* and *dump* command, by adding their tests to the list of dependent tests. With this information, *Latch* can prevent unnecessary or unreliable tests from slowing down the test suite, and delaying actionable feedback.

5.3. End-to-End Testing: Automating Manual Testing on Hardware

Developers of embedded software rely heavily on manual testing of their programs on the targeted hardware. The goal of manual testing is to verify that both the hardware and software of the system work correctly. It is equally important to check that the effects on the environment and the interaction between the hardware and the environment, work as intended. This kind of comprehensive end-to-end testing of embedded systems requires extensive control over the environment and conditions the hardware operates under, such as simulating user interactions, or controlling the input for sensors. These requirements account in large part for the ubiquity of manual testing, since they make automation of testing much more difficult.

Latch allows tests to control the behavior of the environment with local actions, and the behavior of the software under test through debugging instructions. This enables developers to script automated tests that correspond with manual testing scenarios.

When performing end-to-end testing on the hardware, whether manual or automated, things outside the control of the system can go wrong and cause the test to fail even though no part of the software under test is at fault. Such failures are often rare and non-deterministic, leading to flaky tests. The built-in detection and reporting of flaky tests in *Latch* is therefore important for end-to-end testing scenarios with the hardware.

Example: Testing MQTT Primitives. The WARDuino virtual machine has a callback handling system that is used to implement different asynchronous IoT protocols [37], such as primitives for the MQTT protocol. Since the correct implementation of such protocols is crucial for applications, we need to test it extensively. Unfortunately, the public WARDuino project currently has no automated tests for these components, especially since they require interaction with the device to be tested. The following example illustrates how *Latch* can be used to write end-to-end tests for both the callback system and the MQTT primitives. The example wants to verify the following two requirements:

1. After the subscribe primitive is called, the callback function should be registered for the correct topic in the virtual machine's callback system.

2. When an MQTT message is received the correct callback function should be called.

To test this functionality, we use a minimal program that subscribes on a single MQTT topic, and through a callback writes all messages it receives to the serial bus. An AssemblyScript implementation is shown in Listing 23.

```
1  function echo(topic: string, payload: string): void {
2      print(payload);
3  }
4
5  export function main(): void {
6      // ...
7      mqtt_init("broker.hivemq.com", 1883);
8      mqtt_subscribe("echo", echo);
9      // ...
10 }
```

Listing 23: Tiny MQTT program used to regression test the callback handling system in WARDuino.

The code in Listing 23, leaves out the code that connects to the Wi-Fi network, and checks the connection with the server whenever the program is idle. The example instead focuses on the three main things the program needs to do for the end-to-end test. It configures the MQTT server on Line 7, and subscribes to the *echo* topic on Line 8 with the callback function defined on Line 1. The scenario in Listing 24 uses the program to test the callback system and MQTT primitives of the WARDuino virtual machine on real hardware.

```
1  const test: Test = { // MQTT test
2    title: "Test MQTT primitives",
3    program: "mqtt.ts",
```

```
  dependencies: [testWiFi],
  steps: [
    { title: "Start program",
      instruction: Command.invoke("main", []) },
    { title: "CHECK: callback function registered",
      instruction: Command.dumpCallbackMapping,
      expect: [{
        callbacks: (state, mapping) => mapping.some((map) => map["echo"].length > 0)}] },
    { title: "Set breakpoint at *echo* callback",
      instruction: Command.setBreakpoint(breakpointAtFunction("echo")) },
    { title: "Send MQTT message and await breakpoint hit",
      instruction: Actions.messageAndWait() },
    { title: "CHECK: entered callback function",
      instruction: Command.dump,
      expect: [{mode: Mode.PAUSE}, {func: "echo"}] }
  ]
};
```

Listing 24: Test for the callback handling system in WARDuino, showing multiple steps and a custom assertion.

Many hardware-specific tests require the environment to behave in a controlled way. *Latch* makes no assumptions about the hardware and environment used for testing. Instead, the test specification language offers the ability to define local actions, through which the tester in the framework can manipulate and control the environment, both real and simulated.

The first step of the scenario invokes the main function, and the second step checks whether the echo callback was correctly registered in WARDuino's internal callback mapping. In the third step, the scenario sets a breakpoint at the callback function, so in the next step it can check if the callback is indeed called whenever an MQTT message is sent. To this end, the fourth step tells the tester to perform a local action. In the example, the *messageAndWait* function will send a message to the MQTT broker and wait until the testee reports that a breakpoint is hit. Once its promise resolves, we know a breakpoint is hit, and the final step double-checks whether we are indeed in the right function. When the promise is rejected, however, the action is marked as failing before continuing the scenario. This fifth step retrieves a dump of the current virtual machine state, and checks that WARDuino is paused and the current function corresponds to the *echo* callback function.

6. Performance Evaluation

The goal of *Latch* is to allow large-scale testing of IoT software on microcontrollers, and to enable users to write a versatile range of tests. The framework is open-source and available on GitHub.[3] The testing scenarios in the previous section illustrate the versatility of *Latch* to implement many testing strategies. Sections 3 and 4 show how managed testing works, and what the *Latch* framework does to overcome the three challenges outlined in Section 2. In this section we provide empirical evidence to support our research question:

Question Is the managed testing approach, where tests are split into steps, sufficient for executing large-scale tests with *Latch*?

6.1. Test suites

To answer the question of performance, we execute a number of tests suites with *Latch* on an ESP32-WROVER IE and measure the runtime overhead compared to executing the same suites on a laptop. The test suites include the unit and debugging test suites presented in Section 5, and an additional test suite

[3]The framework and all test suite used in this section can be found on our GitHub repository, along with the artifacts generated for this evaluation, which are published as releases: `https://github.com/TOPLLab/latch`

which is more computationally intensive.[3] We chose these three types of test suites in order to have a wide range of tests that are unique in different aspects. The specification test suites from Section 5.1 are structurally identical, but test very different aspects of computer programs, ranging from memory manipulation, to control flow. The suites also represent a very common test pattern, unit testing through single function invocation, which is ubiquitous in many modern testing practices. The debugging test suite on the other hand, does not limit itself to just the invoke command, but uses the entire range of *Latch* commands in its tests, which also contain multiple steps. The computing test suite is structurally similar to the specification test suites, but is computationally more intensive, with steps that generally take at least an order of magnitude longer to perform.

Large Unit Test Suites. We use the WARDuino specification test suites as found in the public repository of the virtual machine, which we presented in Section 5.1. The collection contains 10,213 total tests across 25 test suites. The tests cover the operations on the numerical values, both integer and floating point, which are the only types of values in WebAssembly. The *copy*, *load*, *align*, and *address* categories test the WebAssembly memory, while the *local tee*, *local set*, *local get*, *nop*, *return*, *call indirect*, and *call* categories test stack manipulation. The remaining tests verify the structured control flow of WebAssembly. During the evaluation we used the default scheduling algorithm in *Latch*, and ran the test suites on a single remote testee.

The developers of the WARDuino virtual machine use simulation to test against the WebAssembly specification. However, the simulation ignores important hardware limitations. For instance, the memory of the simulated hardware is only limited by the amount of memory available to the host machine. Furthermore, to execute the specification tests, the WARDuino developers extended the simulator with a dedicated parsing library to parse the test suite written in S-expressions. This parsing library is too big to be run on the ESP32 and the S-expressions from the test suite alone, take up 713 KB of memory. This is already more than twice the size of the microcontroller's memory, without including the WARDuino virtual machine, the parsing library, and the infrastructure to run the test suite. This means, that the WARDuino developers cannot currently test on the microcontrollers they target. However, when comparing the outcome of this approach with the output of the *Latch* version, we found no differences, giving us confidence in the soundness of our framework. To assess the performance of *Latch*, we measure the overhead of executing the *Latch* test suites on a microcontroller compared with current practices, i.e. using a simulator.

WARDuino Debugger Test Suite. While the different specification test suites, test very different aspects, their structure are similar. We therefore include the debugger test suite outlined in Section 5, as an example that is not a traditional unit test suite. Rather than exclusively using the invoke command, this suite uses all commands available to *Latch*.

Computing Test Suite. As a final example, we include a test suite that unit tests a few simple mathematical operations, calculate the factorial, get the nth number in the fibonacci sequence, find the greatest common divider, and check if a number is prime. Similar to the specification test suites, the computing tests each include a single invoke step. However, while the steps in the other test suites are very fast, taking just a few milliseconds—the steps in this test suite can take several centiseconds.

6.2. Single Device Performance

All test suites are run separately on a Dell XPS 13 laptop using an 11th Gen Intel Core i7-1185G7 and 32 GB RAM memory, and the ESP32-WROVER IE microcontroller operating at a clock frequency of 240 MHz, and with 520 KiB SRAM, 4 MB SPI flash and 8 MB PSRAM. Each run starts by initializing the WARDuino instance, in the case of the microcontroller this entails flashing the entire virtual machine to the device. Whenever the test suites use different programs, they are uploaded with the *upload module* command, which allows *Latch* to update the program under test during a test suite, without needing to flash.

A detailed comparison of the overhead of executing the test suite is shown in Fig. 3. The overhead on microcontroller is shown as relative to the simulator, and is the sample mean taken over 10 runs. *Latch*

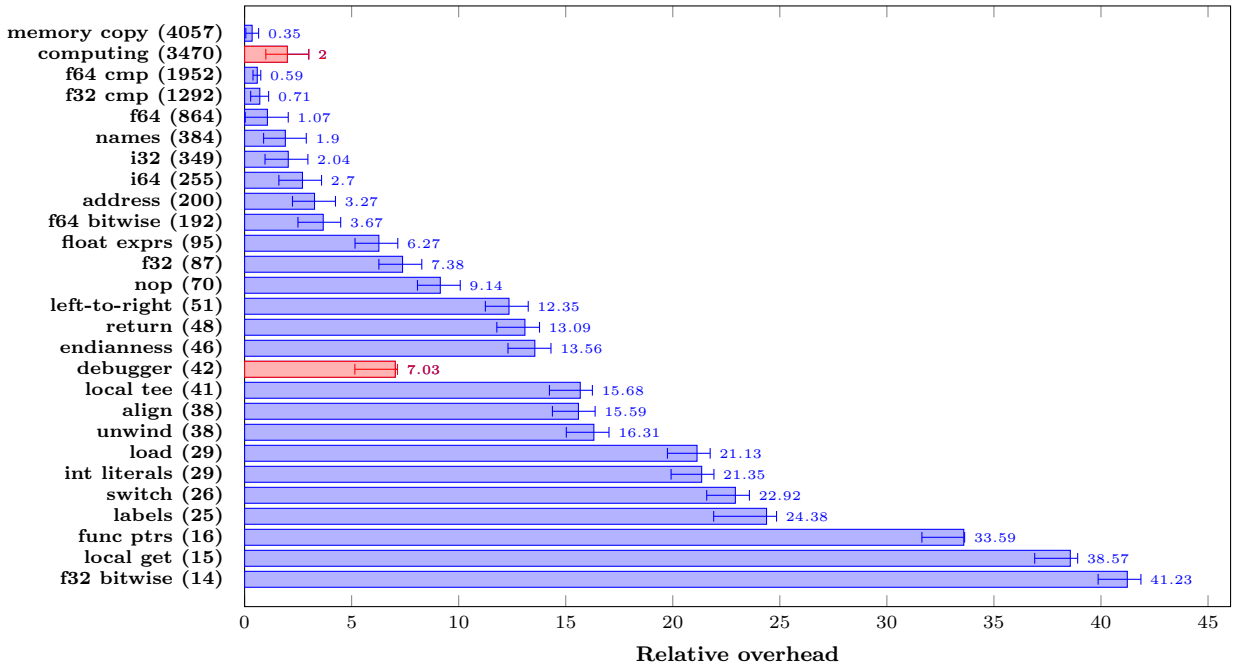

Figure 3: The relative runtime overhead of *Latch*'s WebAssembly specification test suite on hardware compared to a simulator for each test suite. Runtimes are calculated as sample means of 10 runs, and the exact relative overhead is shown next to each bar. The error bars show the confidence interval for the difference between the two means (normalized to the relative overhead) based on the Welch's t-test. The number of steps for each test suite is listed next to its name.

is run in its default mode without the flakiness analysis, where tests are run at most once. Each bar in this graph shows the overhead for executing one test suite on the hardware with *Latch*. The test suites are ordered from most steps, to least. The number of steps are shown next to each name, and the specification tests suites are highlighted in a different color.

All test suites shown in Fig. 3 can be executed with *Latch* on the simulated version of WARDuino in approximately 10 minutes. Executing the same test suites directly on the ESP32, takes around 20 minutes. While the test suites take on average twice as long on the embedded device, the largest of the specification test suites run faster on the microcontroller. This is counterintuitive, but can be explained by the nature of the test suites. The steps in the specification test suites, are very simple tasks that are performed too quickly for any difference to be observed between the two devices, The overhead therefore becomes dominated by the communication, not the actual instructions themselves. The way the TypeScript framework handles the interprocess communication is evidently slower than the serial communication with the microcontroller over USB-C. However, the flashing at the start remains much slower than starting a new process on the laptop, therefore the overhead of the specification test suites with the fewest steps, is dominated by the startup phase instead. This results in the highest overhead overall.

The specification test suites taken separately in Fig. 3, shows that fewer test steps results in higher overhead, because the execution time becomes dominated by the flashing process. This shows how important it is to prevent unnecessary flashing by using the *upload module* command. Conversely, more steps result in lower overhead, because the communication dominates the execution time of the steps. However, the debugger and computing test suites are major outliers, suggesting this is not the full story. For instance, the computing test suite contains 3,470 steps, but has a much higher overhead than the memory copy suite of a similar size. This is due to the steps in the computing test suite being much more computationally intensive, and so much slower. Because the steps take longer to execute, the relative impact of the communication overhead is much lower. The debugger test suite on the other hand, has a much lower overhead than similarly sized specification test suites. This is because, invoke instructions used in the specification test suites, are

	unit testing			debugger			computing			all		
devices	*mean*	*std*	*ovh*	*mean*	*std*	*ovh*	*mean*	*std*	*ovh*	*mean*	*std*	*ovh*
1	908	2.88	200%	34	0.49	665%	299	0.20	235%	1,241	2.30	211%
2	763	2.66	168%	27	0.49	533%	174	0.22	137%	965	1.98	164%
3	751	3.85	165%	28	1.02	547%	133	0.75	105%	912	5.11	155%
4	787	8.05	173%	29	0.81	565%	120	0.73	95%	936	8.28	159%
5	826	4.53	182%	31	0.86	603%	108	1.60	86%	966	4.13	165%

Table 2: The impact of parallelization on the execution time of the test suites in total, and the computing suite taken separately, run on identical ESP32 WROVER E devices, sharing two USB buses, and using the default scheduler. Rows show the arithmetic mean and standard deviation taken over 10 runs, as well as the performance overhead as a percentage of the execution time on a single simulator.

	default	**optimistic**
mean (s)	35.49	34.92
std (s)	0.41	0.62
uploads	12 (44%)	9 (33%)

Table 3: The impact of the default and optimistic scheduler on the debugging suite with dependencies, run on an ESP32 WROVER IE. Rows show the arithmetic mean and standard deviation taken over 10 runs; and the number of uploads used as an absolute number, and as a percentage of tests in the suite.

quite slow compared to most of the other *Latch* commands, used in the debugger test suite. An entire user-defined function is run, in contrast to the step and dump command, which run a single instruction, or only send data. While the differences in structure among the test suites, reveals how many factors impact the performance of *Latch*, the results for the suites are roughly inline with each other. The results show that *Latch* performs well for our use-case of very large test suites of many small unit tests, which are very common in regression testing and continuous integration.

6.3. Multi-device performance

Executing all the test suites on a single microcontroller takes twice as long as the simulator, or 211% of the simulator's execution time to be precise. Since microcontrollers are cheap and widely available, running tests on multiple devices may mitigate this performance overhead. Therefore, the *Latch* framework supports running test scenarios in parallel for a given test suite. We investigated the impact of parallel execution on the previous test suites, by running them on multiple identical ESP32 WROVER E devices. The results showed that the impact of parallelization depends greatly on the structure of the test suite. We therefore, show the results or our experiments in Table 2, grouped by the type of test suites. For each group, we show the arithmetic mean (*mean*) and standard deviation (*std*) of the execution times in seconds. We also give the performance overhead (*ovh*) as a percentage of the execution time on a single simulator.

Generally, the parallelization has a positive impact on the execution time, and is most effective when the execution time is not dominated by the slow flashing procedure. This is clearly illustrated by the small debugger test suite, which only take on average 34 seconds to execute on a single device. However, flashing takes on average around 20 to 22 seconds. This means that there is not much room for improvement, since the flashing procedure is not parallelized. In fact, due to the shared USB bus between devices, the flashing procedure becomes slower when using more devices. This can be observed in the results for the unit testing suites. While less dominating, the flashing procedure still impacts the execution time of these suites significantly. The results in Table 2 show that the average execution time for the unit testing suites initially improves with more devices, but at four or more devices the performance starts to degrade. This is due to an increasingly slow flashing procedure, which at four or five devices can take up to 30 seconds.

By contrast, the execution time of the computing test suite is impacted far less by the flashing procedure. Here adding more than three devices still has a positive impact on the execution time. In fact, starting at four devices, the suite executes faster than on a single simulator. Using five devices, the suite takes 86% percent of the time it takes a single simulator, lowering the overhead from the initial 235% with 149%.

Because the computing test suite takes much less time than the unit testing suites, this performance gain is not as visible in the overall results. However, with the optimal three devices, the overhead for the entire collection of test suites can be reduced from 211% to 155%, a decrease of 56%.

6.4. Impact of Scheduling Algorithms

In the previous experiments, the test scenarios contained no user-defined dependencies. This means that the execution order of test scenarios are the same for both scheduling algorithms. To examine the impact of the scheduling algorithms, we ran an adjusted version of the debugging test suite with dependencies, which contains 27 test scenarios, each using one of five distinct programs. Dependent tests in this adjusted test suite always use the same program.

Table 3 shows the impact of the default and optimistic scheduler on the debugging suite with dependencies. The average execution times for both scheduling algorithms are similar, but for the default scheduler the arithmetic mean over 10 runs is 573 ms lower. This can be explained by the need for the optimistic scheduler to reupload the program under test more often, as shown in the table. The default scheduler uploads the program 12 times, while the optimistic scheduler gets closer to the minimum of 5, by uploading 9 times.

6.5. Threats to Validity

The performance evaluation tries to answer the question at the start of this section, by showing that the managed testing approach, where tests are split into steps, is sufficient for executing large-scale tests with *Latch*. The evaluation includes three different experiments, which use a large collection of test suites.

Internal validity The performance results are affected by many factors, since the benchmarks are run on two devices, the framework on the laptop and the tests on the microcontroller. The communication between the two is an important factor in the runtime performance, and may be influenced by, e.g., the operating system of either device, the configuration of the microcontroller, or the hardware (serial connection) itself. This is especially true for the parallel test runs, where the communication between the devices can start competing with each other on the serial bus. However, we believe given the size and number of repetitions, the performance figures are illustrative for the overall performance of managed testing with *Latch*.

Additionally, the latency of serial communication, and the flashing speed of the devices, are the most important factors for the performance results. Since the performance measurements are taken in a controlled environment, we were able to keep these factors constant. Therefore, we have high confidence in the internal validity of the results.

External validity The biggest threat to external validity, is that the test suites used here, may not be representative for the typical workloads of microcontrollers. This would mean, that the runtime results do not generalize to other microcontroller software test suites. We believe we have mitigated this threat sufficiently, since the specification, computing and debugger test suites are very different structurally, yet present very similar runtime performance. Moreover, the *invoke* instruction, used in the specification test suite, is one of the more expensive operations in *Latch*, since user defined functions may take very long to execute. This is a strong indication that the runtime performance results do generalize, for typical microcontroller workloads.

Construct validity In this evaluation, we use the runtime performance as an indirect measure for the *usability* of *Latch*. Likewise, the variety of the test suites is used as a measure and illustration of usability. Therefore, the construct validity is similarly threated by the representativeness of the test suites.

We believe this is largely mitigated by the extensive specification test suites. These suites use standard unit tests that invoke a function and check its results. It is no coincidence that these kinds of unit tests are so widely spread in test-driven development. They are an excellent way of testing that can be applied to almost any piece of software, regardless of its structure or programming paradigm. This also

holds for microcontroller software. Furthermore, the specification test suites are quite heterogeneous, since the categories test wildly different aspects of the virtual machine—from among others, control flow, arithmetic, stack manipulation, and memory access. For these reasons, we believe that this test suite is able to give a representative evaluation of *Latch*'s performance, while also showing the versatility of the framework.

6.6. Summary

In conclusion, we believe that our evaluation shows that the *Latch* framework and its managed testing approach present a realistic answer to our research question. The framework is able to automatically execute large-scale test suites on constrained devices with good performance, considering the limited processing power of the constrained devices. *Latch* performs the best for our most important use-case, test suites with high numbers of small unit tests for the same software under test. On the other hand, the performance overhead is highest when *Latch* needs to upload new software frequently. The *Latch* prototype has initial support for parallel execution on multiple constrained devices which can help mitigate this overhead, especially when many long-running tests can be uploaded simultaneously to different devices. For our computation test suite, we are able to match and slightly exceed the performance of a simulator, with only five devices working in parallel.

7. Related Work

Common software development practices such as regression testing, continuous integration, and test driven development, are much harder to adopt when working with microcontrollers. This is in large part due to the need to test on the physical hardware, specifically microcontrollers. There are very few solutions for single-target testing of software on microcontrollers. Ztest [54], Unity [56, 67], and ArduinoTest [47] are traditional unit testing frameworks for specific microcontroller architectures. Unfortunately, these frameworks do little to overcome the resource-constraints of microcontrollers themselves, and provide only the most standard unit testing functionality without any tailored solutions for testing on hardware. However, when testing on microcontrollers in this way, the test scenarios often rely on very specific hardware interactions as illustrated by our examples in Section 5. *Latch* addresses this lacuna with its novel testing methodology based on debugging methods. We are not aware of any testing framework that provides an alternative solution.

In this section, we will discuss the differences between *Latch* and the few exiting unit testing frameworks for microcontrollers further. In this paper we have proposed a new way of testing on microcontrollers individually, but IoT systems are often tested as a whole in industry. While this kind of testing answers an entirely different set of demands than *Latch*, we do give a brief overview of these approaches here, for completeness. Similarly, testing plays a large role in general software development. As a result, a wide range of research topics are related to the *Latch* framework, of which not all have been previously applied to IoT. In the remainder of this section, we discuss *holistic IoT testing*, other *unit testing frameworks* broadly, *remote testing*, *scriptable debugging*, *test environments* for IoT programs, *device farms* for mobile applications, *conditional testing*, *test prioritization and selection*, and *flaky tests*. Wherever possible, we include examples from IoT or microcontroller settings.

Unit Testing Frameworks. Constrained devices are still programmed primarily in low-level language such as C and C++. Many traditional unit testing frameworks are available for these languages, such as Google Test [19], Boost.Test [7], CUTE [26], and bandit [6]. There are a handful of frameworks targeting microcontrollers explicitly, such as Unity [56, 67] and ArduinoTest [47]. These work analogous to other unit testing frameworks, but are small enough to run on some constrained devices. While preferable over manual testing, these frameworks require the tests suites to be very small, since they are compiled and run along with the framework in their entirety on the device. In contrast, *Latch* allows arbitrarily large test suites.

Remote Testing. *Latch*'s managed testing is adjacent to remote testing, but with some important differences. Remote testing is not a novel idea, for instance Jard et al. [27] argued in 1999 that local synchronous tests can be translated to remote asynchronous tests without losing any testing power. Remote testing has mostly been used to test distributed systems [70].

The RobotFramework [60] for instance, is a large testing framework that supports remote testing via an RPC interface offering a transparent distribution model. As argued in many papers "distribution transparency is a myth that is both misleading and dangerous" [68, 39, 20]. The Latch framework takes into account these lessons and offers the test engineer a testing framework with inherit timeouts and support for flaky tests, going well beyond the RobotFramework.

Some examples of remote testing frameworks can also be found for constrained devices. The popular PlatformIO project [55], uses the Unity framework [67] for remote testing. However, it works significantly different from how *Latch* executes large test suites. While *Latch* allows arbitrarily large test suites by executing tests step-by-step, Unity does not address the memory constraints of the target devices as it compiles and uploads test suites as one monolithic executable. The framework also does not provide the debugger-like scripts (with custom actions) supported by Latch that enable the automation of standard hardware tests.

Holistic IoT Testing. Existing tools for IoT testing focus largely on testing networked systems of many devices holistically [57, 30], rather than the more common approach where components are tested selectively. Holistic testing of networked systems are by and large incompatible with many of the common development practices; such as test driven development for instance, which relies on selective testing of single components. Moreover, wholesale testing of heterogeneous system is very difficult, so many testing tools instead focus on monitoring to try and detect errors [11, 63]. The few real testing frameworks available, tend to provide testing as a service [33]. While holistic testing makes sense for IoT applications in industry, the approach makes far less sense for more consumer-oriented applications, such as smart home devices. Besides, developers cannot trust that end-to-end testing on such a high level, is enough to test IoT systems thoroughly. Neither does it lend itself well to test-driven development, as testing can only take place with a fully operational system. Therefore, there is a real need for selective—rather than holistic—testing of IoT software on microcontrollers. This is much easier with the single target testing in the style provided by *Latch*.

Scriptable Debugging. *Latch*'s scriptable debugger-like hardware tests are inspired by scriptable debugging, which has been used in many other domains [42]. Scriptable debugging refers to all debugging techniques that can be controlled by developers through a programming language or similar tools such as regular expressions. Programmable debugging goes back to the early eighties, with many of the early proposals, such as Dispel [29] and Dalek [49], exploring variations on the concept of breakpoints. Recent work on a scriptable debugger API for Pharo [12], exposes a wide variety of advanced debugging operations, and allows developers to solve many challenging debugging scenarios through automated scripts. We are not aware of any framework which also applies the idea of scriptable debugging to testing in the context of constrained hardware.

Test Environments. A popular research topic in the domain of IoT testing, are heterogeneous test environments [8], where software can be distributed to nodes which are connected via a controlled network. This solution focuses on the challenging heterogeneity of IoT systems, and does not take into account the constraints the limited memory puts on the test suite size. Most test environments are virtual, and emulate the entire IoT environment [59, 48, 64].

While, simulators are widely used for testing Internet of Things systems [8], they can never capture all the aspects of real hardware [61, 32]. For example, bugs caused by mistakes in interrupt handling, incomplete or wrong configuration, and concurrency faults [41] are typically not simulated. Because accurate hardware emulation is difficult, modern simulators often incorporate parts of the hardware under test, as is the case for *hardware-in-the-loop simulations* [45]. Similarly, some test environments do allow hardware to be integrated into their test environments, but still fundamentally rely on virtualization [3, 31]. There are far fewer works that look into full hardware test environments [1, 9, 18]. Using these large test environments can give more control to the developer to change various aspects of the nodes and network, such as packet loss, latency, and so on. However, setting up such large and often complex systems is complicated and time-consuming, for that reason they are often provided as a service [33, 4]. Subsequently, the test environments confine users to the specific choices in hardware, virtualization, and network technologies made by the service. While these test environments reduce the overhead of setting up a testing lab, they do not fundamentally help developers overcome the hardware limitations faced when executing large test suites.

Device Farms. These test environments are sometimes called testing farms or device farms in case they use real hardware, and are a popular approach for testing mobile applications [25, 15]. Curiously, testing on devices seems much more prevalent in the field of testing mobile applications [34]. We believe this might be because mobile devices have far more memory than the embedded devices targeted by *Latch*, and therefore have no problem running large test suites. This strengthens our view that testing on constrained hardware presents a worthwhile research direction. However, the existing device farms heavily target mobile devices, and again limit users to the chosen technologies and hardware.

Conditional testing. Dependencies in *Latch* can be viewed as conditional skips for tests, where a test is skipped if any of the scenarios it depends on fail. Conditional skips have been around for some time in unit testing frameworks, such as the pytest framework for the Python language [35], and the JUnit framework for Java [2]. Pytest includes a *skipif* annotation which takes a boolean expression as its argument. In JUnit developers can use the *Assume* class, which provides a set of methods for conditional execution of tests. Modern frameworks targeting constrained devices [56, 47] do not support conditional tests.

Test prioritization and selection. Another purpose of the dependencies in the test description language, are to determine the order tests are run in. Research on software testing has recently increased its attention to test prioritization and test selection [51]. These techniques can also be applied to testing IoT systems [43], where they are particularly useful since they can reduce large test suites to the most important tests, and help prioritize tests in such a way that regression tests fail as early as possible. An interesting line of future research could focus on integrating these techniques in *Latch*.

Flaky Tests. Flaky tests represent an active domain of research [53], which focuses on three problems: detecting flaky tests, finding root causes, and fixing flaky tests [71]. The first step is to detect which tests are flaky. A popular approach is to look at the code coverage of tests [71, 5]. Once a flaky test is found, the next step is to find the root cause of the flaky test. This is a considerably harder problem, which is still being actively worked on [36]. Alternatively, some research looks into automatically fixing, or preventing, flaky tests [62]. All these techniques, from detection to fixing, are developed with the ultimate goal of mitigating and preventing flaky tests. In contrast, *Latch* focuses on providing a simple way of detecting and measuring the number of flaky tests in a test suite run. When evaluating *Latch*, we encountered flaky tests only rarely, but we believe that further research is warranted to assess the degree in which testing on constrained devices can cause flaky tests, and how existing techniques can mitigate them.

8. Conclusion

Testing is an essential part of the software development cycle which is currently very challenging on constrained devices. The limited memory and processing power of these constrained devices restrict the size of the test suite and makes testing slow, impeding a fast feedback loop. Moreover, due to the non-deterministic and unpredictable environment, tests can become flaky.

In this paper, we answered the question of how to design and implement a testing framework for automatically running large-scale versatile tests on constrained systems. We introduce our novel testing framework *Latch* (Large-scale Automated Testing on Constrained Hardware), which needed to overcome three challenges; the memory constraints, processing constraints, and the timeouts and flaky tests. In essence, *Latch* splits test suites into small test instructions which are sent by a managing tester to a managed testee (constrained device). Because the constrained device receives the test instructions incrementally from the tester, it does not need to maintain the whole test suite in memory. By using an unconstrained tester to manage the constrained devices and the test suites, *Latch* is able to overcome the memory constraints.

Our testing framework further allows programmers to indicate the dependencies between related tests. This dependency information is used by *Latch* to skip tests that depend on previously failing tests, thus resulting in a faster feedback loop and helping the framework overcome the processing constraints of microcontrollers. On top of that *Latch* addresses the issue of timeouts and flaky tests, by including an analysis mode that provides feedback on timeouts and the flakiness of tests. Finally, the framework uses a novel approach of debugging-like instructions to allows developers to automate manual testing on hardware.

To demonstrate the efficacy and versatility of *Latch*, we showcased three use-cases, each pertaining to one stratum of the testing pyramid. The first use-case exemplifies unit testing, and showcases how we

implemented a large suite of unit tests in *Latch* for a WebAssembly virtual machine intended for constrained devices. This test suite consists of 10,213 unit tests for a virtual machine running on a small ESP32 microcontroller. The second use-case illustrates integration testing of the instrumentation API in *Latch*. The third use-case highlights how the *Latch* test specification language allows programmers to write debugging-like testing scripts to test more elaborate testing scenarios, that mimic common manual testing tasks. Benchmarks show that the overhead of the testing framework is within expectation, roughly matching the performance difference between the constrained hardware and using a simulator on a workstation. Our test-cases shows that the testing framework is expressive, reliable, and reasonably fast, making it suitable to run large test suites on constrained devices.

Acknowledgements

Tom Lauwaerts was supported by a project from the Research Foundation Flanders (FWO) with file number G030320N, and Stefan Marr was supported by a grant from the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

References

[1] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. 2015. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, New York, NY, USA, 459–464. https://doi.org/10.1109/WF-IoT.2015.7389098

[2] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. 2023. JUnit 5 User Guide. Retrieved January 10, 2023 from https://junit.org/junit5/docs/current/user-guide/

[3] Ilja Behnke, Lauritz Thamsen, and Odej Kao. 2019. HéCtor: A Framework for Testing IoT Applications Across Heterogeneous Edge and Cloud Testbeds. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (Auckland, New Zealand) *(UCC '19 Companion)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/3368235.3368832

[4] Jossekin Beilharz, Philipp Wiesner, Arne Boockmeyer, Lukas Pirl, Dirk Friedenberger, Florian Brokhausen, Ilja Behnke, Andreas Polze, and Lauritz Thamsen. 2021. Continuously Testing Distributed IoT Systems: An Overview Of The State Of The Art. In *Service-Oriented Computing – ICSOC 2021 Workshops: AIOps, STRAPS, AI-PA and Satellite Events, Dubai, United Arab Emirates, November 22–25, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 336–350. https://doi.org/10.1007/978-3-031-14135-5_30

[5] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 433–444. https://doi.org/10.1145/3180155.3180164

[6] Stephan Beyer and Joakim Karlsson. 2023. bandit. Retrieved February 15, 2023 from http://banditcpp.github.io/bandit/

[7] Boost.Test team. 2023. What is Boost.Test? Retrieved February 15, 2023 from https://github.com/boostorg/test

[8] Miroslav Bures, Matej Klima, Vaclav Rechtberger, Xavier Bellekens, Christos Tachtatzis, Robert Atkinson, and Bestoun S. Ahmed. 2020. Interoperability and Integration Testing Methods for IoT Systems: A Systematic Mapping Study. In *Software Engineering and Formal Methods*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham, 93–112.

[9] Clément Burin des Rosiers, Guillaume Chelius, Eric Fleury, Antoine Fraboulet, Antoine Gallais, Nathalie Mitton, and Thomas Noël. 2012. SensLAB. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, Thanasis Korakis, Hongbin Li, Phuoc Tran-Gia, and Hong-Shik Park (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 239–254.

[10] Mike Cohn. 2009. *Succeeding with Agile: Software Development Using Scrum* (1th. ed.). Addison-Wesley Professional, Boston, MA, USA.

[11] Datadog. 2024. End to End Testing Automation. https://www.datadoghq.com/synthetics/end-to-end-testing-automation/.

[12] Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega, and Stéphane Ducasse. 2019. Sindarin: A Versatile Scripting API for the Pharo Debugger. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 67–79. https://doi.org/10.1145/3359619.3359745

[13] Espressif Systems. 2023. Unit Testing in ESP32. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/unit-tests.html

[14] Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (Un)Covered Parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation* (Tokyo, Japan) *(ICST)*. IEEE, New York, NY, USA, 230–240. https://doi.org/10.1109/ICST.2017.28

[15] Mattia Fazzini and Alessandro Orso. 2020. Managing App Testing Device Clouds: Issues and Opportunities. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 1257–1259.
[16] Fermyon Technologies, Inc. 2023. WebAssembly Language Support Matrix. Retrieved January 10, 2023 from https://www.fermyon.com/wasm-languages/webassembly-language-support
[17] David Flanagan. 2020. *JavaScript: The Definitive Guide* (7th. ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.
[18] Alexander Gluhak, Srdjan Krco, Michele Nati, Dennis Pfisterer, Nathalie Mitton, and Tahiry Razafindralambo. 2011. A survey on facilities for experimental internet of things research. *IEEE Communications Magazine* 49, 11 (2011), 58–67. https://doi.org/10.1109/MCOM.2011.6069710
[19] GoogleTest. 2023. GoogleTest User's Guide. Retrieved February 15, 2023 from https://google.github.io/googletest/
[20] R. Guerraoui. 1999. What object-oriented distributed programming does not have to be, and what it may be. *Informatik* (1999). http://infoscience.epfl.ch/record/83554
[21] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 27–36. https://doi.org/10.1145/3357390.3361029
[22] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (jun 2017), 185–200. https://doi.org/10.1145/3140587.3062363
[23] Johan Haleby. [n. d.]. Awaitility. http://www.awaitility.org/.
[24] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, New York, NY, USA, 1–23. https://doi.org/10.1109/SCAM.2018.00009
[25] Jun-fei Huang. 2014. AppACTS: Mobile App Automated Compatibility Testing Service. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, New York, NY, USA, 85–90. https://doi.org/10.1109/MobileCloud.2014.13
[26] IFS Institut für Software. 2023. CUTE. Retrieved February 15, 2023 from https://cute-test.com/
[27] Claude Jard, Thierry Jéron, Lénaïck Tanguy, and César Viho. 1999. *Remote testing can be as powerful as local testing*. Springer US, Boston, MA, 25–40. https://doi.org/10.1007/978-0-387-35578-8_2
[28] JetBrains s.r.o. 2023. WebStorm. Retrieved August 28, 2023 from https://www.jetbrains.com/webstorm/
[29] Mark Scott Johnson. 1981. Dispel: A run-time debugging language. *Computer Languages* 6, 2 (1981), 79–94. https://doi.org/10.1016/0096-0551(81)90068-0
[30] Teemu Kanstrén, Jukka Mäkelä, and Pekka Karhula. 2018. Architectures and Experiences in Testing IoT Communications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 98–103. https://doi.org/10.1109/ICSTW.2018.00034
[31] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 15, 15 pages.
[32] Muhammad Zahid Khan, Bob Askwith, Faycal Bouhafs, and Muhammad Asim. 2011. Limitations of Simulation Tools for Large-Scale Wireless Sensor Networks. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications* (Biopolis, Singapore). IEEE, New York, NY, USA, 820–825. https://doi.org/10.1109/WAINA.2011.59
[33] Hiun Kim, Abbas Ahmad, Jaeyoung Hwang, Hamza Baqa, Franck Le Gall, Miguel Angel Reina Ortega, and JaeSeung Song. 2018. IoT-TaaS: Towards a Prospective IoT Testing Framework. *IEEE Access* 6 (2018), 15480–15493. https://doi.org/10.1109/ACCESS.2018.2802489
[34] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* 68, 1 (2019), 45–66. https://doi.org/10.1109/TR.2018.2865733
[35] Holger Krekel and pytest-dev team. 2023. pytest: helps you write better programs. Retrieved January 10, 2023 from https://docs.pytest.org/en/7.2.x/
[36] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 101–111. https://doi.org/10.1145/3293882.3330570
[37] Tom Lauwaerts, Carlos Rojas Castillo, Robbert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix. 2022. Event-Based Out-of-Place Debugging. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 85–97. https://doi.org/10.1145/3546918.3546920
[38] Tom Lauwaerts, Robbert Gurdeep Singh, and Christophe Scholliers. 2024. WARDuino: An Embedded WebAssembly Virtual Machine. *Journal of Computer Languages* (Feb. 2024), 101268. https://doi.org/10.1016/j.cola.2024.101268
[39] Doug Lea. 1997. Design for open systems in Java. In *Coordination Languages and Models*, David Garlan and Daniel Le Métayer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–45.
[40] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning about JavaScript Promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (oct 2017), 24 pages. https://doi.org/10.1145/3133910

[41] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering* (Madrid, ES) *(ICSE)*. IEEE, New York, NY, USA, 460–472. https://doi.org/10.1109/ICSE43902.2021.00051

[42] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. 2007. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering* 14, 1 (01 Mar 2007), 59–86. https://doi.org/10.1007/s10515-006-0003-z

[43] Noha Medhat, Sherin M. Moussa, Nagwa Lotfy Badr, and Mohamed F. Tolba. 2020. A Framework for Continuous Regression and Integration Testing in IoT Systems Based on Deep Learning and Search-Based Techniques. *IEEE Access* 8 (2020), 215716–215726. https://doi.org/10.1109/ACCESS.2020.3039931

[44] Microsoft. 2023. The TypeScript Handbook. Retrieved February 7, 2023 from https://www.typescriptlang.org/docs/handbook/intro.html

[45] Franc Mihalič, Mitja Truntič, and Alenka Hren. 2022. Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges. *Electronics* 11, 15 (2022). https://doi.org/10.3390/electronics11152462

[46] Vadym Mukhin, Yaroslav Kornaga, Yurii Bazaka, Ievgen Krylov, Andrii Barabash, Alla Yakovleva, and Oleg Mukhin. 2021. The Testing Mechanism for Software and Services Based on Mike Cohn's Testing Pyramid Modification. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Vol. 1. 589–595. https://doi.org/10.1109/IDAACS53288.2021.9660999

[47] Matthew Murdoch. 2023. ArduinoUnit. Retrieved February 15, 2023 from https://github.com/mmurdoch/arduinounit

[48] Fotis Nikolaidis, Manolis Marazakis, and Angelos Bilas. 2021. IOTier: A Virtual Testbed to evaluate systems for IoT environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, New York, NY, USA, 676–683. https://doi.org/10.1109/CCGrid51090.2021.00081

[49] Ronald A Olsson, Richard H Crawford, and W Wilson Ho. 1990. Dalek: A GNU, Improved Programmable Debugger.. In *USENIX Technical Conference*, Vol. 90. The USENIX Association, Berkeley, CA, USA, 221–231.

[50] OpenJS Foundation. [n. d.]. Mocha - the Fun, Simple, Flexible JavaScript Test Framework. https://mochajs.org/.

[51] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. 2021. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering* 27, 2 (14 dec 2021), 29. https://doi.org/10.1007/s10664-021-10066-6

[52] Daniel Parker. 2015. *JavaScript with Promises* (1st ed.). O'Reilly Media, Inc.

[53] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (oct 2021), 74 pages. https://doi.org/10.1145/3476105

[54] Yuval Peress, Anas Nashif, Manoel Brunnen, Henrik Brix Andersen, Andrei Emeltchenko, Aaron E. Massey, Marti Bolivar, and Ivan Herrera Olivares. 2024. Test Framework — Zephyr Project Documentation. https://docs.zephyrproject.org/latest/develop/test/ztest.html.

[55] PlatformIO. 2023. Unit Testing. Retrieved February 8, 2023 from https://docs.platformio.org/en/stable/advanced/unit-testing/index.html

[56] PlatformIO. 2023. Unity. Retrieved February 15, 2023 from https://docs.platformio.org/en/latest/advanced/unit-testing/frameworks/unity.html#unity

[57] Svitlana Popereshnyak, Olha Suprun, Oleh Suprun, and Tadeusz Wieckowski. 2018. IoT application testing features based on the modelling network. In *2018 XIV-th International Conference on Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. 127–131. https://doi.org/10.1109/MEMSTECH.2018.8365717

[58] Python Software Foundation. [n. d.]. Doctest — Test Interactive Python Examples. https://docs.python.org/3/library/doctest.html.

[59] Brian Ramprasad, Marios Fokaefs, Joydeep Mukherjee, and Marin Litoiu. 2019. EMU-IoT - A Virtual Internet of Things Lab. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, New York, NY, USA, 73–83. https://doi.org/10.1109/ICAC.2019.00019

[60] Robot Framework Foundation. 2023. Robot Framework. Retrieved August, 2023 from https://robotframework.org/

[61] Tamás Roska. 1990. Limitations and complexity of digital hardware simulators used for large-scale analogue circuit and system dynamics. *International Journal of Circuit Theory and Applications* 18, 1 (1990), 11–21. https://doi.org/10.1002/cta.4490180104 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cta.4490180104

[62] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. IFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 545–555. https://doi.org/10.1145/3338906.3338925

[63] SolarWinds Worldwide, LLC. 2024. AppOptics – APM and Infrastructure Tool | SolarWinds AppOptics. https://www.solarwinds.com/appoptics.

[64] Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. 2020. Fogify: A Fog Computing Emulation Framework. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, New York, NY, USA, 42–54. https://doi.org/10.1109/SEC50012.2020.00011

[65] The AssemblyScript Project. 2023. AssemblyScript. Retrieved January 10, 2023 from https://www.assemblyscript.org/

[66] The JUnit Team. [n. d.]. JUnit 5. https://junit.org/junit5/.

[67] Mark VanderVoord, Mike Karlesky, and Greg Williams. 2015. UNITY. https://www.throwtheswitch.org/unity

[68] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. 1996. A Note on Distributed Computing. In *Mobile Object Systems*. https://api.semanticscholar.org/CorpusID:2390403

[69] Haochen Xie. 2017. Principles, Patterns, and Techniques for Designing and Implementing Practical Fluent Interfaces in Java. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming,*

Languages, and Applications: Software for Humanity (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 45–47. https://doi.org/10.1145/3135932.3135948

[70] Yizheng Yao and Yingxu Wang. 2005. A framework for testing distributed software components. In *Canadian Conference on Electrical and Computer Engineering, 2005.* IEEE, New York, NY, USA, 1566–1569. https://doi.org/10.1109/CCECE.2005.1557280

[71] Behrouz Zolfaghari, Reza M. Parizi, Gautam Srivastava, and Yoseph Hailemariam. 2021. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience* 51, 5 (2021), 851–867. https://doi.org/10.1002/spe.2929 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2929