Multi-Mode Debugging for FRP-Based Embedded Systems

Yugo Otani
Institute of Science Tokyo
Tokyo, Japan
yugo@psg.comp.isct.ac.jp

Sosuke Moriguchi
Institute of Science Tokyo
Tokyo, Japan
chiguri@comp.isct.ac.jp

Takuo Watanabe
Institute of Science Tokyo
Tokyo, Japan
takuo@comp.isct.ac.jp

Abstract

Emfrp is a functional reactive programming (FRP) language designed for small-scale embedded systems. Time-varying values are the primary abstraction mechanism in FRP and enable concise descriptions of reactive behavior. In practice, however, Emfrp programs are compiled into C and combined with platform-dependent input/output components written in C or C++. Consequently, developers must debug the resulting mixed C/C++ program using conventional debuggers such as GDB, even though the application logic is written in Emfrp. This situation creates an abstraction gap between the source-level FRP program and the executable system.

This paper presents a multi-mode debugging framework for Emfrp-based embedded applications. The framework supports debugging at the level of Emfrp abstractions while also allowing inspection of platform-specific C/C++ I/O code. Our approach uses a source code mapping technique that relates Emfrp constructs to corresponding locations in the compiled program. A case study on an ESP32 microcontroller using representative debugging scenarios demonstrates improved debugging efficiency.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Functional languages*; • **Computer systems organization** → *Embedded systems.*

Keywords: Debugging, Functional Reactive Programming, Embedded Systems, Microcontrollers

ACM Reference Format:
Yugo Otani, Sosuke Moriguchi, and Takuo Watanabe. 2026. Multi-Mode Debugging for FRP-Based Embedded Systems. In *Proceedings of 4th ACM International Workshop on Future Debugging Techniques (DEBT '26)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEBT '26, Brussels, Belgium
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Debugging reactive systems written in functional reactive programming (FRP) can be challenging, particularly in embedded environments. FRP programs describe system behavior as relationships among *time-varying values* (aka *signals*) [1, 2], that is, values that evolve over time and can conceptually be viewed as functions of time. By representing reactive behavior as data dependencies among such values rather than explicit event-driven control flow, FRP enables developers to describe reactive systems at a higher level of abstraction.

However, this abstraction also introduces difficulties during debugging. Errors in FRP programs are often embedded in temporal dependencies among reactive values, making it difficult to determine which update of which value at what time caused an unexpected behavior. This challenge becomes even more pronounced in embedded environments, where runtime observation and instrumentation are often limited.

Emfrp [9] is an FRP language designed for small-scale embedded systems. The language provides built-in support for time-varying values and allows their current values to be referenced directly in expressions. To support resource-constrained platforms, Emfrp prohibits recursive functions and recursive data structures, enabling static determination of memory usage.

The Emfrp compiler translates Emfrp programs into C source code. The generated code implements the FRP runtime that periodically updates time-varying values according to their definitions. Device-dependent input/output functionality is implemented separately by developers in C or C++. The generated C code and the developer-written I/O code are then compiled together using a C compiler for the target platform to produce an executable for the embedded device.

Currently, debugging Emfrp programs relies on conventional embedded debugging techniques such as JTAG-based debugging with tools like GDB. In this setting, developers must debug the generated C program directly, forcing them to manually relate high-level Emfrp constructs to low-level C code. This mismatch between abstraction levels creates a significant obstacle to effective debugging.

To address this problem, we propose a multi-mode debugging framework for Emfrp-based embedded applications. In this framework, a mode corresponds to the level of abstraction at which debugging is performed. The framework

supports two modes: debugging at the level of Emfrp source code and debugging at the level of platform-specific C/C++ I/O code. By bridging these levels, developers can investigate system behavior in terms of Emfrp abstractions while still accessing the underlying implementation when necessary. Based on this framework, we implement Emdb, a debugger that provides Emfrp-aware debugging operations and integrates them with an existing C debugger.

The contributions of this paper are as follows:

- FRP-aware debugging operations tailored to Emfrp programs.
- Source mapping and command translation connecting Emfrp code with generated C code and existing C debuggers.
- Evaluation on an ESP32 platform demonstrating improved debugging efficiency.

2 Motivation

2.1 Emfrp

Emfrp [9] is a statically typed, purely functional reactive programming (FRP) language designed for small-scale embedded systems, typically targeting microcontrollers. The language is designed so that programs can run even on highly resource-constrained environments with only a few kilobytes of RAM and flash memory. In typical deployments, target platforms provide from a few kilobytes to several tens of kilobytes of RAM and up to a few megabytes of flash memory. To support such environments, Emfrp programs are designed to have a small memory footprint and to allow their runtime memory usage to be determined statically. The Emfrp compiler translates Emfrp source code into C code, which is compiled together with device-specific input/output code written in C or C++ using a C compiler for the target platform to produce an executable for the embedded device.

In Emfrp, time-varying values are represented as *nodes*, which form the basic abstraction of FRP computation. Each node represents the current value of a time-varying quantity that evolves over time. Inputs from the external environment, such as sensor readings or button states, are represented as *input nodes*, while outputs to external devices are represented as *output nodes*. An Emfrp program is organized as a module consisting of a header that declares input and output nodes followed by a sequence of type definitions and node definitions. Nodes are defined using the following syntax:

node init[*c*] *name* = *e*

Here, *name* denotes the node being defined, optional **init** [*c*] specifies the initial value *c* of the node, and *e* is an expression that computes the current value of the node.

Listing 1 shows an example Emfrp program that toggles an LED each time a button is double-clicked. To avoid unintended repeated activations, the program ignores button inputs for a short cooldown period after a double-click is detected. In this module, the node `btn`, which represents the current button state, and the node `nowMs`, which represents the elapsed time since system startup in milliseconds, are declared as input nodes. The node `led`, representing the LED state, is declared as an output node. The remainder of the program defines intermediate nodes that detect button presses, determine whether a click or double-click has occurred, and maintain the cooldown state.

```
1  module DoubleClick
2     in btn(False) : Bool,  # button
3        nowMs: Int          # msec from reset
4     out led : Bool
5
6  node init[False] led =
7    if isDouble then !led@last else led@last
8
9  type Opt[a] = Some(a) | None  # option type
10
11 # Time when the button press started
12 node init[None] pressTime =
13    if !inCooldown && !btn@last && btn
14    then Some(nowMs)
15    else pressTime@last
16
17 # Whether a click is detected
18 # (press duration <= 250 ms)
19 node clicked = {
20    up = btn@last && !btn
21    pressTime@last of:
22       Some(t) ->
23         !inCooldown && up && nowMs - t <= 250
24       None -> False
25 }
26
27 # Time when a click was detected
28 node init[None] clickTime =
29   if clicked then Some(nowMs)
30              else clickTime@last
31
32 # Whether a double click is detected
33 # (second press within 500 ms after a click)
34 node init[False] isDouble : Bool = {
35    dwn = !btn@last && btn
36    clickTime@last of:
37       Some(t) -> dwn && nowMs - t <= 500
38       None -> False
39 }
40
41 # Time when the cooldown period ends
42 # after a double click is detected
43 node init[None] cooldownUntil =
44    if isDouble then Some(nowMs + 500)
45                else cooldownUntil@last
46
47 # Whether the system is currently in the
48 # cooldown period
49 node inCooldown =  cooldownUntil@last of:
50    None -> False
51    Some(t) -> nowMs < t
```

Listing 1. DoubleClick Module

Emfrp provides the keyword **@last** to access the value of a node at the previous moment. For example, the expression `!btn@last && btn` becomes true only when the button transitions from released to pressed, corresponding to the

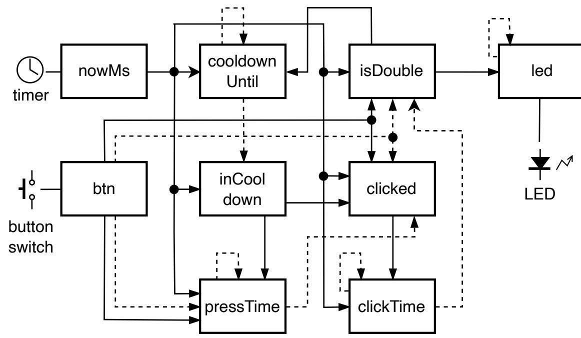

Figure 1. Dependency Graph of the Nodes in `DoubleClick`

rising edge of the button signal. Nodes can also maintain state using this mechanism. For instance, the definition of led toggles the LED state whenever a double-click is detected while preserving the previous state otherwise.

Figure 1 shows the dependency graph of the nodes in the program of Listing 1. In this graph, each vertex represents a node (i.e., a time-varying value) in the Emfrp program, and each directed edge represents a dependency between nodes. A solid edge indicates a dependency through a direct reference, while a dashed edge represents a dependency through a reference using **`@last`**. The graph formed by solid edges must be acyclic; that is, the direct dependency graph must be a directed acyclic graph (DAG). The Emfrp compiler[1] performs a topological sort of this DAG to obtain a linear ordering of nodes and generates code that updates node values in that order. In Emfrp, one complete update of this ordered list of nodes is called an iteration.

2.2 Debugging Emfrp Programs

In practice, debugging Emfrp programs follows the standard workflow used for embedded systems. The development PC is connected to the target microcontroller through a debugging interface such as JTAG, and a debugger running on the host PC (e.g., GDB) interacts with the device through a debug server such as OpenOCD[2]. As described in the Section 1, however, the program being debugged is not the original Emfrp program but the C code generated by the Emfrp compiler together with device-specific C/C++ I/O code. As a result, developers must analyze a mixed C/C++ program whose structure differs significantly from the original Emfrp source.

Listing 2 shows a fragment of the generated C code corresponding to the node `clicked` in Listing 1. Although the generated code faithfully implements the semantics of the Emfrp program, its structure differs considerably from the original source. For example, the compiler introduces temporary variables (such as `_tmp004`) and additional control

```
1  static int node_clicked(int nowMs,
2    struct Maybe_Int* pressTime_at_last,
3    int button_at_last, int button, int* output) {
4    int _tmp005;
5    int _tmp004;
6    _tmp005 = _anpersand_anpersand(button_at_last,
7        _at__exclamation_(button));
8    if (1) {
9      int pvar3_released = _tmp005;
10     int _tmp006;
11     if (pressTime_at_last->tvalue_id == 0) {
12       int pvar4_t0 =
13         pressTime_at_last->value.Some.member0;
14       _tmp006 =
15         _anpersand_anpersand(pvar3_released,
16         _lt__eq_(_minus_(nowMs, pvar4_t0), 250));
17     }
18     else {
19       _tmp006 = 0;
20     }
21     _tmp004 = _tmp006;
22   }
23   *output = _tmp004;
24   return 1;
25 }
```

Listing 2. Compiled Code of the Node `clicked`

structures for code generation purposes. These artifacts do not appear in the original Emfrp program and make it difficult for developers to identify which conditions determine the value of each node.

Another challenge arises from differences in identifier naming and data representation. Although the compiler attempts to preserve some correspondence between identifiers in Emfrp and those in the generated C code, the names are not always identical and may vary depending on the context in which they are used. Furthermore, user-defined algebraic data types in Emfrp are translated into combinations of C integers, structures, and enumerations. Developers must therefore reconstruct the original Emfrp-level data structures by examining how these low-level representations are accessed and manipulated in the generated C code.

A further complication stems from the execution model of Emfrp programs. As described in the previous subsection, nodes are evaluated in a topologically sorted order of their dependency graph. The keyword **`@last`** allows a node to refer to its value from the previous iteration. At runtime, the system maintains both the current value and the previous value of each node. However, this distinction is not always explicit in the generated C code, making it difficult for developers to understand whether a variable corresponds to a current value or a previous one when debugging at the C level.

These factors significantly increase the cognitive burden of debugging Emfrp programs at the C level. Developers must simultaneously reason about the original Emfrp program, the generated C code, and the runtime representation

[1] https://github.com/psg-titech/emfrp
[2] https://openocd.org

of program state. Consequently, much of the debugging effort is spent reconstructing the correspondence between abstraction levels rather than diagnosing the actual cause of the bug. This situation motivates the need for a debugging framework that allows developers to observe and control program execution directly at the Emfrp level.

3 Bug Analysis and Requirements

3.1 Taxonomy of Bugs in Emfrp Programs

To design an effective debugging framework for Emfrp, it is useful to classify the kinds of bugs that arise in Emfrp-based systems and to identify what forms of observation and control are needed to diagnose them. An Emfrp application can be viewed as consisting of three layers: the *FRP layer*, which is written in Emfrp and implements the main application logic; the *I/O layer*, which is written in C and handles interaction with external devices; and an *intermediate layer*, which contains primitive data types and primitive functions implemented in C. Because these layers differ in both language and role, the kinds of bugs that arise in them, and the debugging support required to diagnose those bugs, also differ. In this paper, we focus primarily on the FRP layer and the I/O layer.

At the FRP layer, bugs can be divided into two broad categories: *time-independent bugs* and *time-dependent bugs*. Time-independent bugs are those that do not essentially depend on references to previous values through `@last`. Typical examples include errors in conditional branches, logical expressions, arithmetic expressions, or argument order. Conceptually, these are similar to ordinary bugs in functional programs: the value computed in a single evaluation step is incorrect. To localize such bugs, a debugger must reveal which branch was taken, the values of local variables, and the values of relevant subexpressions at runtime.

Time-dependent bugs arise from the temporal structure of Emfrp programs with `@last`. Without `@last`, an Emfrp module behaves essentially like a combinatorial logic circuit: its outputs are functions of the current inputs and it cannot express stateful behavior over time. In practice, `@last` is used for at least three purposes: implementing states, detecting moments of change, and breaking dependency cycles. In Listing 1, nodes `cooldownUntil`, `pressTime`, `clickTime`, and `led` use `@last` to retain state across updates; these self-dependencies appear as dashed self-loops in Figure 1. The expression `!btn@last && btn` (lines 13 and 35 in Listing 1) illustrates the second use, namely detecting the instant at which the button is pressed. Other uses of `@last` in the example correspond to resolving cyclic dependencies. In these cases, typical bugs include missing or incorrect conditions, inconsistent initial values, and mistakes in where `@last` is applied. Such bugs are often harder to diagnose because the relevant cause may lie not only in the current expression but also in the relationship between current and previous values.

The I/O layer has a different character. It is written in C and is responsible for interaction with external devices. Although arbitrary C code can be used in this layer, Emfrp assumes that developers do not introduce long blocking operations that prevent the progress of iterations or asynchronous updates that interfere with the FRP layer, for example through interrupts that destructively modify shared state. Typical bugs in this layer include incorrect peripheral configuration or initialization, hardware-dependent mistakes, and timing-related problems such as an inappropriate sampling period for an external device. Timing-related bugs are particularly problematic because suspending execution with a debugger may itself perturb the timing behavior and reduce reproducibility. This suggests that the framework should support not only stop-and-inspect debugging but also forms of observation that do not rely on intrusive suspension.

3.2 Requirements

The observations in the previous subsection lead to several requirements for an Emfrp debugger. First, because the I/O layer and the intermediate layer are written in C, the framework should cooperate with an existing C debugger rather than replace it entirely (**Req-1**). Second, for state observation at the Emfrp level, the framework should display the current execution position in Emfrp source code (Req-2), show Emfrp variables such as nodes and local variables despite identifier transformations in generated code (Req-3), visualize the dependency graph to help developers understand state and state transitions (Req-4), provide stack traces at the level of Emfrp functions rather than C functions (Req-5), and support inspection of expression evaluation (Req-6). Third, for execution control, the framework should support multiple granularities of stepping, including stepping by subexpression, by node update, and by iteration (Req-7), as well as watchpoints for changes in node values (Req-8) and breakpoints at multiple granularities (Req-9). Finally, for bug reproduction, the framework should support input-trace-driven execution (Req-10). Together, these requirements motivate a multi-mode debugging framework that combines Emfrp-level debugging support with integration into existing C-level debugging infrastructure.

4 Design and Implementation of Emdb

Based on the requirements discussed in the previous section, we designed and implemented Emdb, a debugger for Emfrp programs. Figure 2 shows the overall architecture of Emdb and the surrounding debugging environment. Emdb consists of two main components: a frontend provided as an extension for Visual Studio Code (VSCode), and a backend implemented in Python. The frontend accepts user operations such as continue, stop, step execution, and breakpoint

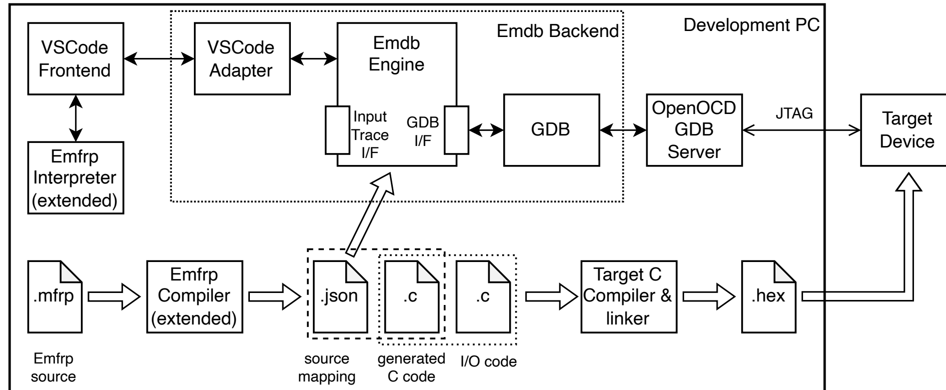

Figure 2. Emdb Architecture

management, and presents the execution state through multiple views. These views include the Emfrp source view, a breakpoint view, node values, a call stack, a dependency graph, a debug console, and an input-trace view. In particular, the source view displays the current execution position in Emfrp source code and supports breakpoints on source locations and subexpressions, while the dependency-graph view visualizes node states and updates them as execution proceeds. The debug console supports both Emfrp-level expression evaluation and low-level C debugger commands.

The backend interprets requests from VSCode and provides debugging functionality at the Emfrp level. Internally, it consists of a VSCode adapter, which translates requests from the frontend into backend API calls, and the Emdb engine, which implements the core debugging logic. The engine also provides an interface for registering input traces in advance when input-trace-driven execution is used. At its foundation, the Emdb engine relies on a GDB interface that controls an existing C debugger. In this sense, Emdb can be viewed as a wrapper that adds Emfrp-aware observation and control on top of a conventional C-level debugger. This design directly addresses Req-1: debugging of the I/O layer and the intermediate layer can be delegated to an existing C debugger, while generic mechanisms such as target control, stop-reason retrieval, and low-level expression evaluation can be reused. In our implementation, we provide standard GDB-interface backends based on GDB/MI and on the Python interface of LLDB, which makes Emdb reusable across multiple targets with little additional effort.

To support source-level debugging of Emfrp programs, we extended the Emfrp compiler so that it generates not only C source code but also a JSON-based source mapping file. This mapping describes the correspondence between the generated C code and the original Emfrp source, and is the basis for reconstructing Emfrp-level execution from C-level debugger events. The mapping contains four kinds of information: (1) the Emfrp element corresponding to each relevant location in generated C code, (2) the correspondence between Emfrp variables and C variables, (3) the correspondence between Emfrp types and C-level representations, and (4) dependency information among nodes. In addition, we extended the Emfrp interpreter so that Emfrp expressions can be evaluated directly inside the debugger. Together, these extensions allow Emdb to present execution state, values, and types in terms of Emfrp rather than generated C.

A central concept in the mapping is the checkpoint. A checkpoint is a C-level execution location, identified at line granularity, that has a meaningful correspondence to an Emfrp-level element such as a syntax element, a top-level FRP execution event, or a function definition. The compiler assigns checkpoints so that transitions among checkpoints in C execution order reproduce the corresponding execution order at the Emfrp level, and so that at most one checkpoint is assigned to each line of generated C code. At the FRP top level, checkpoints are placed on lines corresponding to input processing, output processing, node initialization, and node update function calls. This makes transitions among checkpoints correspond to node-level stepping along the dependency order. Inside node-update functions, additional checkpoints are introduced for expressions. Arithmetic operations, function applications, and other expression constructs are compiled into a form close to three-address code, so that transitions among checkpoints correspond to subexpression-level stepping. More complex constructs such as conditional expressions and pattern matching are also compiled in a way that preserves Emfrp-level execution structure. Using these checkpoints, Emdb observes C-level execution through the GDB interface, tracks transitions between mapped locations, and reconstructs execution in terms of Emfrp semantics.

5 Case Study: Finding a Bug in `DoubleClick`

Listing 1 contains a subtle bug, and in this section we illustrate how Emdb can be used to locate it. The target program, `DoubleClick`, controls an LED using a single button. A click is recognized only when the button is pressed for at most 250 ms, and a double click is recognized when the

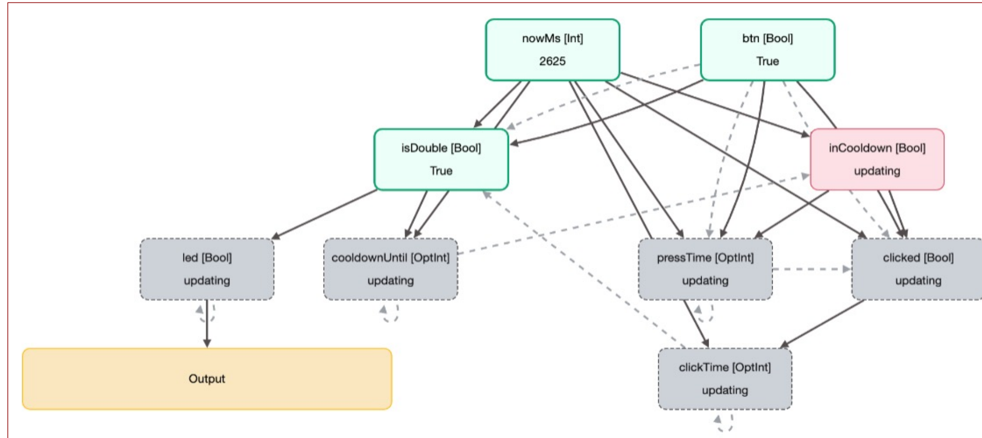

Figure 3. Dependency Graph View of DoubleClick

second click occurs within 500 ms. After a double click is detected, the system is intended to ignore further button inputs for a 500 ms cooldown period. However, the program in Listing 1 contains a bug: if a third click occurs within 500 ms after a double click, the LED is toggled again. We confirmed this behavior on the target device. This example is representative of timing-dependent bugs that are important in embedded-system development and difficult to diagnose with conventional low-level debugging alone.

We use Emdb to debug this behavior as follows. First, Emdb is configured to record an input trace, that is, a sequence of external inputs observed during execution. While this recording feature is enabled, we perform a triple click on the real device and confirm the faulty behavior. Next, we replay the program on the host PC using the recorded trace. This allows us to reproduce the timing-dependent bug deterministically without relying on repeated manual input on the device. To reach the point where the first double click is recognized, we set a conditional breakpoint on the event that node isDouble becomes true. Equivalently, a breakpoint can be placed on the then branch of the if expression in the definition of led. After execution stops there, we set another breakpoint, or watchpoint, on the expression `!button@last && button`, which represents the third button press. Continuing execution then brings the debugger to the third press event.

At this point, we inspect execution while displaying the dependency-graph view and stepping over execution at the granularity of node updates. Figure 3 shows the resulting state. The graph view indicates that the node isDouble becomes True again at the third click, which should not happen if the cooldown mechanism is working correctly. The same figure also shows the current values of nodes such as nowMs and btn, together with the nodes currently being updated, making it easier to understand the propagation of state through the dependency graph. This immediately reveals that the unexpected LED toggle is caused by the incorrect evaluation of isDouble.

To determine why `isDouble` becomes true, we restart execution using the same recorded input trace and stop again at the same point. Emdb supports deterministic re-execution from a previously saved input trace, so the same execution state can be reproduced without recreating the interaction manually. We then step into the definition of `isDouble` and examine the branch selected in the pattern match on `clickTime@last`. The debugger shows that execution reaches the branch corresponding to `Some(t)`, meaning that a previous click time is still stored. At this point, the local variable corresponding to the most recent press event indicates that the button has just been pressed, while the values of `t` and `nowMs` show that the current press still falls within 500 ms of the stored click time. As a result, the condition for `isDouble` evaluates to true even though the system should already be in the cooldown period.

This observation reveals the actual cause of the bug. The previous click time is preserved in `clickTime`, so the third press is still interpreted as occurring within 500 ms of the previous click. In other words, the cooldown mechanism prevents clicked from being updated, but it does not invalidate the previously stored click time. As a result, the old click timestamp remains available to isDouble, and the cooldown is effectively bypassed. To fix this problem, either isDouble must explicitly check `inCooldown`, or `clickTime` must be cleared when a double click is detected. In our implementation, we adopt the former solution by adding `!inCooldown` to the `Some(t)` branch of `isDouble`. After this modification, the program behaves correctly, which we confirmed both in replay-based debugging and on the target device.

We discuss the qualitative benefits of Emdb compared with using GDB directly on the generated C code. When a conventional C debugger is used, understanding Emfrp-level

state requires the developer to manually reconstruct correspondences between Emfrp nodes or local variables and the identifiers and values that appear in the generated C code. In contrast, Emdb directly presents node values and local variables at the Emfrp level through its variable view. Similarly, C-level debugging requires the user to manually translate Emfrp expressions, identifiers, and types when setting breakpoints, conditional breakpoints, and stepping through execution. Emdb eliminates this burden by providing these operations directly at the Emfrp level. In addition, Emdb supports input-trace-driven execution, which allows developers to replay and modify input sequences without regenerating logs, editing C code, or rebuilding the program. These features substantially reduce the amount of auxiliary work required during debugging. The experiments are conducted using an ESP32 microcontroller board (ESP32 DevKitC).

6 Related Work

Several existing debugging approaches are related to Emdb, but they target different execution models and debugging granularities. Salvaneschi and Mezini proposed Reactive Inspector [8], a debugger for the Scala FRP library REScala [7]. Like Emdb, Reactive Inspector emphasizes visualization of dependency graphs and navigation over them as primary debugging operations. However, the transition model differs from that of Emdb. In REScala, recomputation is triggered only when the value of a dependent signal changes, so graph transitions correspond to propagation caused by value changes. In contrast, Emfrp reevaluates nodes in every iteration regardless of whether dependent values change, making iteration-based traversal of the dependency graph more natural. Moreover, Reactive Inspector does not explicitly provide observation of evaluation inside node definitions, whereas Emdb supports debugging at subexpression granularity.

Perez and Nilsson proposed a debugger for Yampa [5], an FRP library in Haskell. Their approach exploits the property that replaying the same discrete sequence of input events yields the same execution result, allowing past executions to be reproduced by saving and reusing input traces. They also combine this idea with QuickCheck to automatically generate event sequences and search for counterexamples. This replay-based view of debugging is closely related to Emdb's input-trace-driven execution, which likewise supports reproducible debugging of reactive behavior.

Rojas Castillo et al. proposed a language-independent debugging approach based on a debugging-oriented control-flow graph (CFG) for WebAssembly [6]. This work is closely related to ours because it also considers WARDuino [3], a WebAssembly runtime for microcontrollers, as a target platform and thus shares our interest in debugging under the constraints of embedded systems. However, although their approach supports multiple source languages, it is primarily aimed at procedural languages compiled to WebAssembly. Emdb, in contrast, targets a pure FRP language with a substantially different execution model, requiring specialized source mappings, stepping semantics, and visualizations tailored to FRP computation.

Iyenghar et al. proposed model-based design-level debugging for embedded systems developed in a model-driven style [4]. Their approach consists of a lightweight target-side monitor and a host-side GUI debugger that visualizes target behavior at the design-model level using runtime traces. The design philosophy is similar to one mode of Emdb, in which execution observed on the target can be reconstructed and inspected on the host. However, their approach assumes UML-based design models and focuses on RTOS-level events, whereas Emdb reconstructs execution at the level of an FRP language. Their discussion of communication interfaces is also relevant: they compare serial communication and JTAG, and report that JTAG provides lower overhead, while buffering strategies can further reduce cost for high-frequency events. These observations are useful when considering communication mechanisms for Emdb.

7 Conclusion and Future Work

This paper presented a multi-mode debugging framework for FRP-based embedded systems and its implementation, Emdb, for the Emfrp language. Our approach addresses the abstraction gap that arises when Emfrp programs are compiled into C and combined with platform-specific C/C++ I/O code, forcing developers to debug a mixed low-level implementation rather than the original FRP program. By integrating Emfrp-level observation and control with an existing C debugger, Emdb enables developers to inspect execution in terms of nodes, dependencies, and temporal behavior while still retaining access to the underlying implementation when necessary.

The case study demonstrates that Emdb can effectively support the diagnosis of timing-dependent bugs that are difficult to locate with conventional C-level debugging alone. Our preliminary evaluation also suggests that Emdb substantially improves debugging practicality for Emfrp programs, while keeping the overhead of several debugging operations sufficiently low for practical use.

There are several directions for future work. First, we plan to conduct a more systematic quantitative evaluation, including controlled comparisons of debugging efficiency. Second, although our current experiments mainly target Espressif ESP32 platforms, we would like to validate the approach on a wider range of embedded targets. Finally, we plan to investigate more unified source-mapping techniques for cross-paradigm debugging, rather than constructing language-specific mappings individually, for example by exploring whether approaches such as CFG-based representations can be extended to FRP and other programming paradigms.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. We are also grateful to Go Suzuki for his insightful suggestions regarding microcontroller-based systems. This work was supported in part by JSPS KAKENHI Grant Numbers JP22K11967 and JP24K14892.

References

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4 (2013), 52:1–52:34. doi:10.1145/2501654.2501666

[2] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*. ACM, 263–273. doi:10.1145/258949.258973

[3] 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 (MPLR '19)*. ACM, 27–36. doi:10.1145/3357390.3361029

[4] Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, Juergen Wuebbelmann, and Michael Uelschen. 2017. *Model-Based Debugging of Embedded Software Systems*. Springer, 107–132. doi:10.1007/978-1-4614-2266-2_5

[5] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proceedings of the ACM on Programming Languages* 1, ICFP (Sep. 2017), 2:1–2:27. doi:10.1145/3110246

[6] Carlos Rojas Castillo, Matteo Marra, and Elisa Gonzalez Boix. 2025. A Control-Flow Graph Approach to Language-Agnostic Debugging for Microcontrollers. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)*. ACM, 38–56. doi:10.1145/3759426.3760979

[7] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *13th International Conference on Modularity (Modularity 2014)*. ACM, 25–36. doi:10.1145/2577080.2577083

[8] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *38th ACM/IEEE International Conference on Software Engineering (ICSE 2016)*. ACM, 796–807. doi:10.1145/2884781.2884815

[9] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems. In *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*. ACM, 36–44. doi:10.1145/2892664.2892670