Towards Enabling Deployment of Lingua Franca on Distributed Embedded Devices

Anirudh Rengarajan
Edward A. Lee, Ed.
Alberto L. Sangiovanni-Vincentelli, Ed.
Marten Lohstroh, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-185
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-185.html

May 19, 2023

Acknowledgement

To my family, friends, and colleagues in the Lingua Franca group, I dedicate this report.

Towards Enabling Deployment of Lingua Franca on Distributed Embedded Devices

Anirudh Rengarajan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee

Edward Ashford Lee
Research Advisor

5/12/23
(Date)

* * * * * * *

DocuSigned by:
Alberto Sangiovanni-Vincentelli
853F4B43F43F48E...

Alberto Sangiovanni-Vincentelli
Second Reader

5/12/2023
(Date)

Abstract

Towards Enabling Deployment of Lingua Franca on Distributed Embedded Devices

by

Anirudh Rengarajan

5th Year Masters in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee

Lingua Franca, a polyglot coordination language, provides deterministic reactive concurrency and time functionality that is preserved even in distributed environments. Most of Lingua Franca's implementation is optimized for computer platforms like MacOS and Linux, and is in its infancy for embedded platforms like Arduino. As such, federated execution, the distributed runtime implementation of Lingua Franca, does not account for hardware constraints in its implementation, such as limits on threads and memory overhead. At the same time, Reactor-C, the runtime library for executing Lingua Franca in C, relies on a level assignment system for determining when reactions can execute in place of storing the entire reaction graph in the runtime library. Though such a level assignment system works for standard Lingua Franca programs, deploying programs over an entire federation can lead to federates (individual programs that communicate with one another) deadlocking unless constraints that were lost when converting from federation to federates, are added. This report first examines how to add support for Arduino and Arduino-compatible boards pointing out the problem posed by the high memory overhead of the Reactor-C runtime. Then it proposes an updated framework for creating the federation in the Reactor-C suite. The framework is based on modular reactor generation and novel approaches to address deficiencies in the level assignment algorithm, including Max Level Allowed to Advance (MLAA) and Total Port Ordering (TPO). MLAA and TPO show promise in the future deployment of federated programs onto low-overhead devices like Arduino.

Contents

Acknowledgments

To start, I want to thank my postdoc advisor Dr. Marten Lohstroh for taking me under his wing early and giving me the opportunity to work on Lingua Franca as both an undergrad and masters student. I appreciate him giving me the chance to explore my passions in Embedded Systems and for providing me valuable projects to work on for Lingua Franca and its C Runtime.

Additionally, I would like to thank my advisor Professor Edward A. Lee for giving me the opportunity to take part in the 5th Year Masters program and extend my time to research and learn here at UC Berkeley. I also want to thank Professor Alberto Sangiovanni-Vincentelli for his insight on embedded system design and scheduling and for taking the time to be the second reader for my report.

I also want to thank Peter Donovan, Erling Jellum, and Shaokai Lin, my fellow lab members, for their mentorship and assistance in facilitating support for Lingua Franca onto low-overhead embedded platforms and optimizing federated execution for Lingua Franca's C target.

Being a part of the Lingua Franca group in the iCyPhy lab has been a highlight of my three years at Berkeley, and I am forever grateful.

Lastly, I want to thank my family: my parents Ravi and Sujatha as well as my sister Subhashree for their unwavering support throughout my life.

To my family, friends, and colleagues in the Lingua Franca group, I dedicate this report.

Chapter 1

Lingua Franca

1.1 Overview

Lingua Franca (LF) [1] is a polyglot coordination language for writing programs in different languages that provides deterministic reactive concurrency and time functionality. The primary languages that Lingua Franca supports include C, C++, Python, TypeScript, and Rust. Lingua Franca's C runtime system (also known as Reactor-C) allows for distributed, concurrent execution of programs that can be theoretically deployed in multiple mediums: from the Cloud to the Edge to bare metal embedded platforms like Zephyr and Arduino [2].

1.2 Reactors

Lingua Franca programs are built on the concept of reactors, independent software components that react to events (inputs, timers, and internal triggers) [3], [4]. These components preserve determinism and concurrency [5] in event-driven reactive systems. Reactors are considered to be the Object Oriented Programming analog of a class in Lingua Franca. We can further break down reactors as follows:

Components

Reactors have triggers in the form of input ports, actions, and timers that are used to queue reactions. Reactors also have local state, parameters, output ports, and an ordered list of reactions.

Reactions

Reactions are an analog of functions inside reactors that are initiated in response to a trigger event. These reactions are required to indicate which inputs they read and which outputs

they write so that all inputs and outputs will have timestamps equal to the event that triggered the reaction.

Composition

Reactors can instantiate other reactors and manage connections between defined reactors that dictate the flow of messages. Outputs of each reactor can be redirected to the inputs of other reactors.

Causality Loops

Under a normal LF program, the flow of reactions across reactors can form a cycle, where downstream reactions can connect back to an immediate upstream reaction, creating a causality loop since precedence relations between relations cannot be satisfied. When this occurs, the LF program will yield an error since it is impossible to execute circular-dependent reactions on the same logical time.

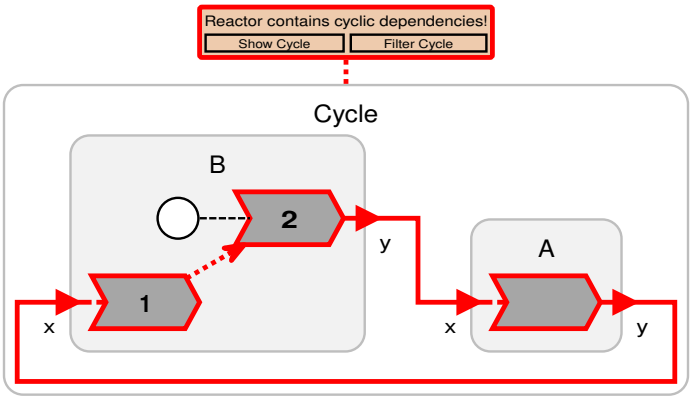

Figure 1.1: Example of a causality loop in a normal LF program failing

To avoid a causality loop, reactions can either be reordered, use physical connections (pass down the current physical time rather than the upstream logical time) or use delayed connections that delay (or advance) logical time for downstream reactions to enforce a proper ordering and to prevent a causality loop.

Events

Triggers (messages between reactors, timers, and actions) are events that come with logical timestamps. These timestamps can trigger downstream reactions at the current logical time. Such events can pass down values as arguments to these reactions.

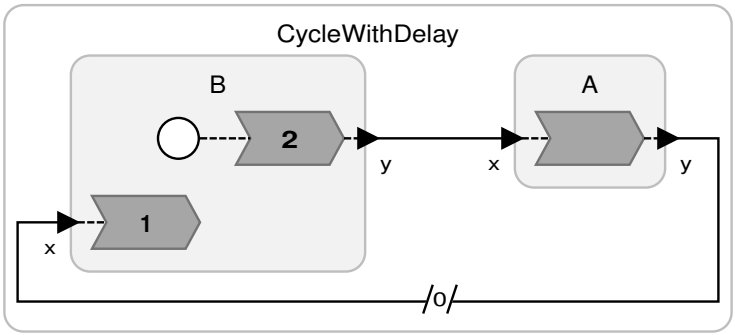

Figure 1.2: Example of breaking a causality loop using a zero-delay, which causes later reactions to occur one microstep after the current logical time.

Mutual Exclusion

Within a reactor, the executions of any two reactions are mutually exclusive. Additionally, when reactions are invoked at the same logical time, they will be invoked in the order of the reactor definition.

Determinism

Given the same input data, the composition of all reactors has the same behavior. We test determinism through a suite of regression tests that determine if the correct behavior is achieved as intended.

Concurrency

In an LF program, a developer must explicitly define dependencies between reactions. When reactions are not dependent on each other, they can run in parallel.

1.3 Time

In Lingua Franca, every event relies upon logical time, where messages are sent and arrive at a logical time instant and reactions to these messages are "logically instantaneous." For every time instant, a reactor's input will either be absent or present with a value. Reactions within a reactor that are connected to the input can be triggered by a present input, by actions, or by timers. When reactions produce outputs, inputs connected to those outputs become present at said logical time instant. When programs start executing, logical time is set to the physical time of the system.

As downstream reactions are triggered, they are executed at the same logical time unless a time delay is used. When reactions in the same reactor are triggered, they must be

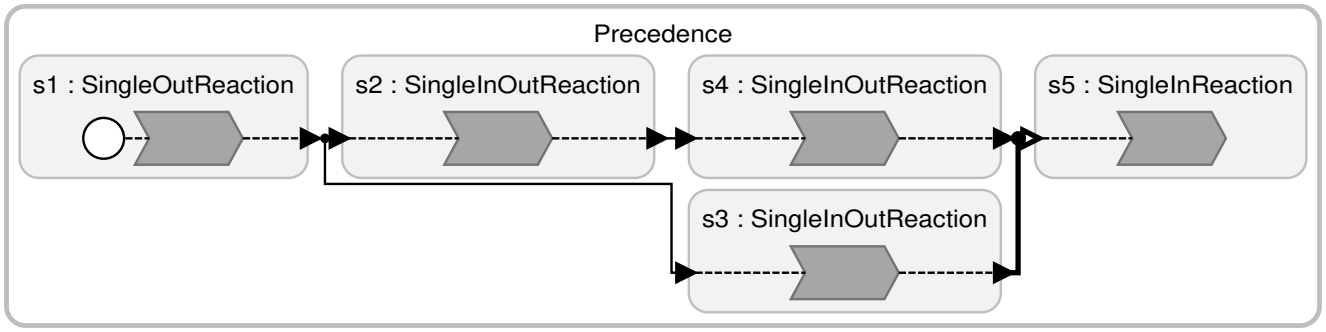

Figure 1.3: Example of a downstream reaction graph. The reaction under the s1 reactor marks the head of the reaction chain. Before each subsequent reaction can occur, the s1 reaction must first execute to preserve determinism. Since reaction s2 (in reactor s2) and reaction s3 (in reactor s3) are not on the same causal chain, such reactions can execute in parallel to leverage concurrency.

executed atomically in the definition order through sequential execution. When reactions are connected upstream, upstream reactions must execute first before their downstream reactions are allowed to occur.

1.4 Federated Execution

Lingua Franca's guarantee of determinism and concurrency is appealing for distributed systems that require predictable coordination of program timings over a network. As such, Lingua Franca programs can be executed in a distributed format for C and TypeScript through a process known as federated execution outlined in [6]. In federated execution, any reactors defined in a federated reactor will generate separate LF programs (called federates) that can be independently compiled. In theory, such federates can be deployed across different mediums from the cloud to the edge, and have a coordination mechanism to preserve time semantics across the network. At its core, a standard LF program should be robust enough to be distributed over a network should a developer request for execution to be federated. Federated execution is in its infancy: we discuss its current implementation and the deficiencies of the Reactor-C runtime's implementation in the next chapter.

Chapter 2

Reactor-C: Lingua Franca's C Runtime

2.1 Overview

When a Lingua Franca program is compiled, depending on what target language is specified, a generated file gets bundled with a reactor runtime library for that particular target language. This section will cover the LF library most commonly used by low-overhead embedded systems like Arduino in Reactor-C.

This library is responsible for managing the Lingua Franca program and interfacing with the generated C file. Reactor-C has support for both threaded and unthreaded implementations, as will be covered in this section.

2.2 Multi-Platform Support

In Reactor-C, there exists a concept of platform-agnostic behavior from a developer perspective since we want to limit the amount of platform-specific code a user has to write to support their LF program. As such, Reactor-C comes bundled with various platform support files to support the following runtime platforms:

- Linux
- MacOS
- Windows
- Arduino
- Zephyr

Each platform support file contains implementations of API functions used by the library to call functions specific to each platform for the appropriate task. Examples of platform-specific functionality include initializing, retrieving hardware time, and managing threads, as well as condition variables, and mutex locks.

2.3 Event and Priority Queues

In both the threaded and unthreaded cases, the overall workflow of a Reactor-C program remains similar. When a program is compiled, the LF Compiler will generate a .c file that contains the C representation of everything in the .LF file. Such information is in the form of global lists and structures of all reactions, triggers, actions, timers, and states. Reactor-C operates on two separate queues: the event queue and reaction queue.

Event Queue

The global event queue is a data structure in Reactor-C sorted by timestamps. When an event is popped off the queue, we retrieve the event with the smallest timestamp (i.e., earlier events come before later events). Events are made up of timers and triggers with time that are used to queue reactions or other events.

Reaction Queue

The reaction queue is a data structure sorted by precedence that controls the order in which reactions are processed at a logical time step. The workflow of the queue is as follows:

Workflow

1. When the LF program starts, timer events are placed on the queue and logical time is initialized to the current physical time of the platform.
2. For every logical time, pop all events (if they exist) from the event queue with identical earliest timestamps and place any triggering reactions onto the reaction queue.
3. Wait until physical time catches up to the earliest timestamp, then advance logical time to that timestamp.
4. Execute reactions off the reaction queue, which may add additional events onto the event queue. Once the queue is empty, go back to the event queue.

For Reactor-C, the reaction queue is made up of a set of binary heaps where each heap represents a set of reactions that have the same level, as will be discussed in this next section.

2.4 Level Assignments

Scheduling of tasks on Embedded Real-Time Systems is a well-documented problem of optimization with regards to how to best allocate workers and task ordering given constraints on ordering and deadlines for tasks [7], [8]. Reactor-C takes an approach to scheduling that lowers memory overhead while still preserving determinism in the order in which reactions (a set of schedulable tasks in LF) are allowed to execute relative to one another.

Unlike other targets, Reactor-C abstracts away the LF reaction dependency graph in its runtime and instead relies on the concept of levels. Levels are a constrained form of priority that indicate whether or not a reaction is permitted to execute. LF's guarantee of determinism requires that reactions at lower levels execute before reactions at higher levels can be allowed to execute. Reactions that exist at the same level are allowed to execute in parallel.

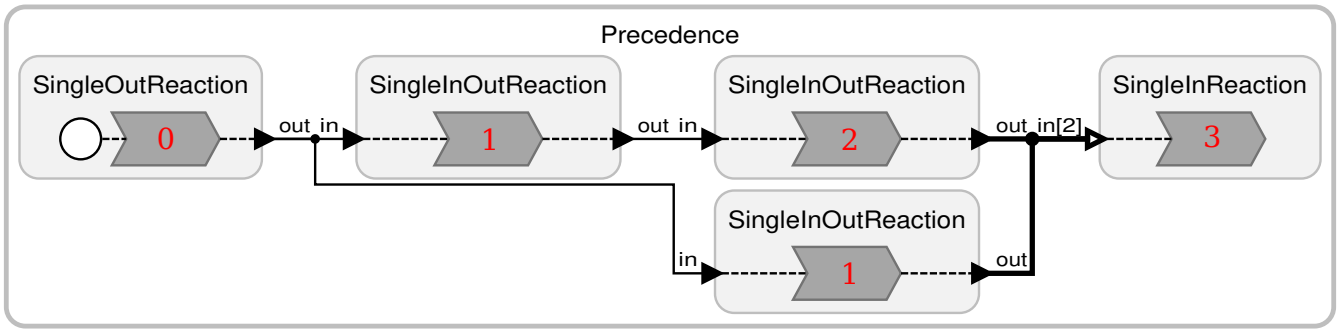

Figure 2.1: Valid Level Assignments in red for the following reaction graph. Reactions with the same levels are allowed to execute concurrently, but previous level reactions must occur before downstream reactions can execute.

The concept of levels is introduced for C programs through the Lingua Franca Compiler. As described in [9], the reaction graph of an LF program is an acyclic precedence graph that fulfills a partial order on scheduling constraints to maintain determinism. At compile time, the Lingua Franca Compiler will use topological sort on the underlying reaction graph using T.C. Hu's level scheduling algorithm [10] to statically assign a level ordering that fulfills constraints based on ordering and priority of reactions. There are an infinite number of possible level assignments, and we must ensure that the level assignment does not create deadlock.

Spurious Dependencies

By abstracting away the reaction graph at runtime, the level assignments create a phenomenon known as "spurious dependencies" where downstream reactions at higher levels

are technically dependent on upstream reactions at lower levels even though they are not directly dependent on one another.

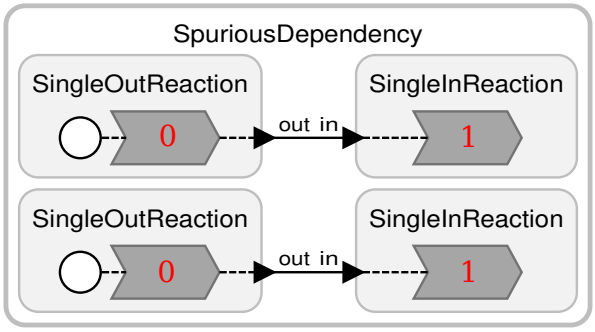

Figure 2.2: Valid Level Assignments in red for the following reaction graph that cause "spurious dependencies." The top right reaction with level 1 is intrinsically dependent on the bottom left reaction with level 0 even though they are not on the same causal chain. We tolerate this since we are not subject to a scenario where we can deadlock.

In general, spurious dependencies are acceptable since we still have the guarantee of determinism so long as upstream reactions are guaranteed to execute without blocking indefinitely. However, this constraint leads to problems when attempting to do level assignments in a federated infrastructure.

2.5 Federated Execution in Reactor-C

Federated Execution as described in Chapter 1 is Lingua Franca's implementation of a distributed system where separate LF programs can be coordinated over a network to perform tasks with predictable timing while maintaining determinism and concurrency where possible. There currently exists support for federated execution on C and TypeScript; however, we cover the C implementation alone as a part of this report. The primary difference between federated and normal execution resides in the handling of network ports and coordination mechanisms for timing between federates (individual LF programs that communicate with one another). The process of converting an LF federation into individual LF federates is discussed at length in Chapter 4.

Centralized Coordination

Centralized coordination in Lingua Franca is an extension of High Level Architecture (HLA) [11]. When coordination is set to centralized (default), all advancements in time for each federate must be regulated by the RTI, a runtime infrastructure program for MacOS and Linux that serves as a coordinator for all messages in the network. In a federate, if events

occur at a certain logical time t, the federate must get approval from the RTI to advance its local logical time to t, which it gains so long as the federate has received every message previously up to logical time t. Once a time advance is granted to federate A, the RTI cannot provide a time advance to federate B above time t until federate A receives its message. Once the RTI delivers A its message, it will provide a Tag Advance Grant to federate B.

Decentralized Coordination

Decentralized coordination in Lingua Franca is an extension of PTIDES [12], [13], a framework that allows for federates to execute mostly independent of a coordinator and without a communication bottleneck. When coordination is set to decentralized, the RTI will only handle startup, clock synchronization, and shutdown. Otherwise, each federate will initiate dedicated sockets with each other to bypass the RTI and will not use the RTI to advance logical time. In its place, each federate will have a safe-to-process (STP) offset where a federate is allowed to advance its logical time to value t when its physical clock is at least t + STP [14].

Network Ports

Communication between federates works similarly to connections between reactors in the sense of ports. When a federate sends a message to another federate, the output port of one federate makes a connection with the input port of another federate. Since each federate is a stand-alone program, we create abstractions of ports by code-generating reactions that handle network inputs and outputs in place of port connections between reactors.

Input Ports with connections from a federate can be classified under three states:

1. Unknown
2. Absent Message
3. Present Message with a value

When a new logical time step begins, all of its network ports are by default unknown. As network messages come into each federate, the status of each port is updated depending on the message that comes in. Absent messages indicate that the connecting federate was not able to provide a value to the current federate by the requested logical time. Messages that have a value are considered as present and are shuttled through the federate to be processed.

Every reaction in a federate has an intrinsic dependency on any upstream network ports it relies on. For a reaction in a federate to execute, it must make sure the port status is not "unknown", whether it be absent or present with a value. If any of its upstream port statuses are unknown, the reaction will need to be blocked, but if all of its upstream port statuses are present or absent, then it can safely be executed. We also discuss the process of how we currently block certain reactions in Chapter 4.

Workflow

When we initialize a federated program on Reactor-C, we first spawn a thread to connect with the RTI and listen for messages coming from the RTI (in centralized communication) or spawn threads to connect to each federate (in decentralized communication) via sockets. Each of these thread(s) will listen over the network and process network messages as they come in. Certain network messages will be port to port (i.e., one federate sending a message to another federate). When such messages arrive, a logical action will trigger a reaction that processes the message and sends it to the appropriate destination input port within the federate. On the other side, output reactions shuttle messages from the federate to the RTI (in centralized coordination) or directly to another federate (in decentralized coordination). At its core, the workflow is well-defined but has increasing orders of complexity as described in Chapters 4 and 5.

Chapter 3

Low-Overhead Cyber-Physical System Support

3.1 Arduino

Arduino is an open-source hardware/software company that designs and fabricates printed circuit board micro-controllers and kits [15]. Such boards have onboard digital and analog I/O pins alongside standardized serial communication interfaces. All Arduino micro-controllers are programmed in C/C++ and are deployable across an array of different board core families listed here:

Core	Architecture	Memory	RTOS?	Common Boards?
AVR	8-bit AVR	Low	No	Uno/Nano
megaAVR	8-bit AVR	High	No	Uno WiFi Rev2
SAM	32-bit ARM Cortex-M3	High	No	Due
SAMD	32-bit ARM Cortex-M0+	High	No	Zero/MKR1000
Mbed	Nordic nRF52840 Cortex-M4	High	Yes	Nano 33 BLE

Arduino programs by design are low-overhead and simplistic due to memory constraints, with most boards having no operating system and most libraries serving basic procedures such as interfacing with sensors and actuators. Arduino programs also rely on a global event loop, which serializes tasks and processing of events, leading to limitations in concurrency. Certain programming languages already exist to augment Arduino's capabilities such as immutable data structures and automated memory management [16].

Lingua Franca's inherent modularity in reactors/reactions and robust timing standards lends itself useful for deployment on bare metal embedded systems like the Arduino family. As such, Lingua Franca now supports an Arduino target based on the Reactor-C runtime. Lingua Franca adds a level of concurrency and determinism to the constrained system of

Arduino without the explicit need for an RTOS. The high-performance deployment of Lingua Franca on embedded devices in place of an RTOS has been well documented for the Zephyr platform [17], and we seek to provide the same robust support for Arduino.

Arduino CLI

Arduino-CLI is a command line tool by the Arduino team that provides sketch building, board detection, upload, and Arduino-compatible board management.[18] Lingua Franca Arduino programs use Arduino-CLI as a build dependency to compile LF-created Arduino sketches and to flash said sketches onto connected boards.

Bare Metal Consideration

Embedded systems without an operating system are known as "bare-metal." Within the Arduino framework, there is a subset of compilation known as bare metal programming where we bypass Arduino CLI altogether and instead directly flash the board with pure C code. Under this system, compilation of Lingua Franca programs is less stringent since we are no longer required to follow function signature requirements of setup and loop and have less overhead by removing the Arduino stack. However, Arduino-CLI provides robust features in terms of pre-made libraries for hardware components like sensors and actuators. Going purely bare-metal would prevent developers from easily being able to integrate Arduino libraries into their LF programs, and as such, we made the choice to fully integrate with Arduino-CLI.

3.2 Lingua Franca Integration

Arduino programs can be broken up into two components: sketches and libraries. Libraries are written in C/C++ and are filled with functions that can be referenced by sketches. Sketches (also known as .ino files) are required to define two functions: setup and loop. Setup is run at the start of an Arduino program and loop is run repeatedly after setup.

Developers can program their Arduino boards with Lingua Franca generated code that integrates with arduino-cli. An example of the differences between the common Arduino program Blink and the LF counterpart for Blink is shown in Figure 3.1.

Unlike Arduino which relies on a global loop in serialized order, Reactor-C schedules reactions in precedence order directly defined by a developer, allowing for more fine-grained control.

Platform Attributes

As indicated in Figure 3.1, we indicate our intention to create an Arduino program by using the C target with a platform value equal to Arduino. The platform attribute can

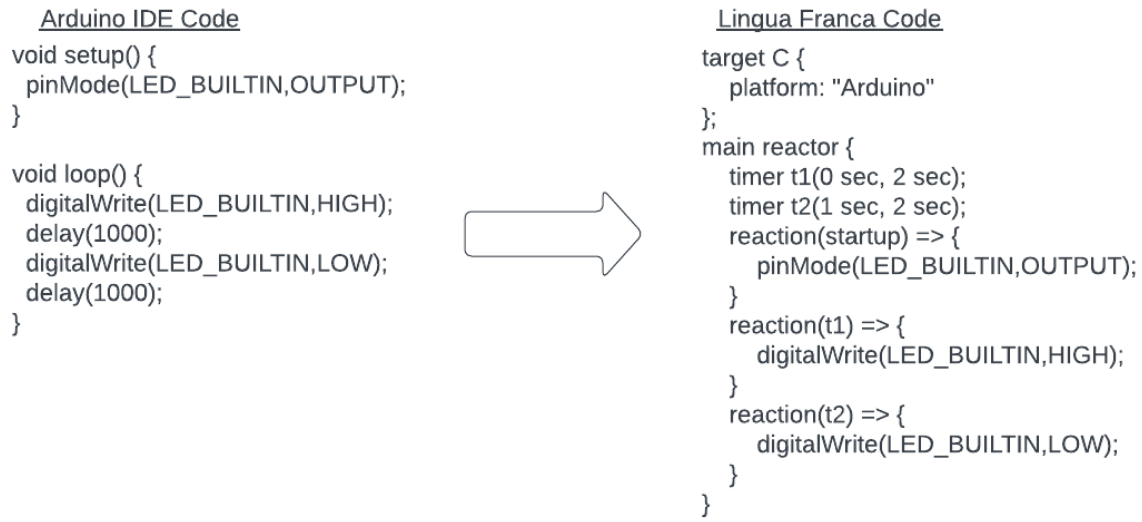

Figure 3.1: Example of the conversion of a serialized Blink program in Arduino to its analog program in LF-Arduino code. Rather than having a global loop, we define reactions offset by separate timers to achieve the same effect.

be overloaded for more fine-grained control over how compilation will work as shown in Figure 3.2.

All Lingua Franca generated Arduino programs will initialize Arduino's Serial Communication Interface (for network communication and print buffers) with a tunable baud-rate in our generated code. Board is an attribute indicating the Fully Qualified Board Name of what Arduino flavor to compile our LF code for, with the above example compiling for the Arduino Uno board. The port indicates the USB location of where to flash the compiled code onto if flash is specified to be true.

Replacement of Setup and Loop

Lingua Franca is made to support loops in the form of timers and other actions and setup in the form of a startup reaction for the main reactor. As such, we code generate the setup function to call Reactor-C's main function. The developer then has fine-grained control over defining reactors and reactions.

Target Declaration

```
target C {
    platform: {
        name: "Arduino",
        board: "arduino:avr:uno",
        port: "/dev/ttyd2",
        flash: true,
        baud-rate: 9600
    }
};
```

Figure 3.2: Example of an overloaded platform declaration with attributes name, board, port, flash, and baud-rate.

Reactor-C as an Arduino Library

To facilitate support with the Arduino-CLI, we treat our Reactor-C runtime as a type of Arduino library that we can import and use from our generated sketch. To ensure compatibility, when we code generate our LF code into C code, we treat our generated file as an ino sketch. Arduino-CLI then sees the sketch and compiles the generated file before linking it with the compiled Reactor-C library.

Relative Library File Compilation

Formally, Arduino Libraries are designed to be header files that can be included by sketches rather than standalone .h and .c files that are compiled separately using a build-tool like CMake. Arduino-CLI does provide a way to recursively compile every file located in a special src directory under the same parent directory as the generated sketch file. During the linking phase, it takes every compile file and links them with the compiled sketch to make the flashable program.

Since we do not have access to CMake in Arduino-CLI, we cannot use the normal Lingua Franca C target build system. Instead, we offload compilation to Arduino-CLI by placing Reactor-C code in the special src directory. Arduino-CLI does not support specification of include directories on compilation, so we post-process this copied Reactor-C code to convert all include directives to relative paths as shown below before compiling.

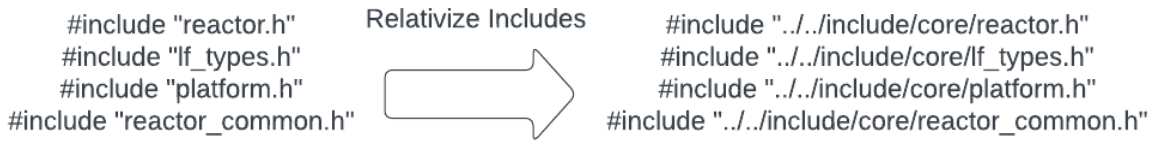

Figure 3.3: Example of the conversion of all includes to relative paths to account for limitations in the Arduino-CLI.

3.3 Converting Time Standards

Arduino-supported Boards run on a 32-bit hardware clock with microsecond granularity that is automatically started when the board is powered up. Lingua Franca on the other hand relies internally on a 64-bit time standard with nanosecond granularity. With the work of Erling Jellum, we found there to be two viable solutions to merge these incompatible timing standards together:

1. Add support for microsecond granularity and 32-bit times in Reactor-C. Such a system would forego the process of converting times from 32 to 64-bit on retrieval from the hardware clock and could provide more fine-grained time control. However, this would require excess maintenance due to different variable signatures and would cause issues on the priority queue in instances of overflows after 35 minutes (signed 32-bit overflow of microsecond timers).

2. Store two variables for time: one to track the current hardware time and one to track overflow. When retrieving the hardware time, we check this time against the stored current time. If it's greater than the current time, we set the current time to this new time. Otherwise, we know the internal hardware time has overflowed, and we increment the overflow counter before setting the current time to the new time. When we need to retrieve the current time for the event and reaction queue, we concatenate the bits of these variables together to create a 64-bit value that can be converted to nanosecond granularity. Such a system would make 8-bit CPU Arduino Cores less efficient since they can only process 8 bits at a time for arithmetic but would keep the rest of Reactor-C's timing standards the same.

To keep our 64-bit, nanosecond granularity consistent across all Reactor-C, we opted to implement the second solution.

3.4 Unthreaded vs. Threaded Runtime Support

Arduino is unique as it has a wide range of boards from mostly bare-metal AVR and ARM Cortex Boards to Arduino BLE/IoT boards with ARM Mbed RTOS support. As such, we need to include flexibility in Reactor-C to account for unthreaded and threaded support.

Unthreaded Runtime

Lingua Franca handles much of the back-end infrastructure needed for the event and reaction queues independent of the platform. From a platform standpoint, there are a few API functions that each platform support file must implement to add that platform's support for Reactor-C.

```
extern void lf_initialize_clock(void);
extern int lf_clock_gettime(instant_t* t);
extern int lf_sleep(interval_t sleep_duration);
extern int lf_sleep_until_locked(instant_t wakeup_time);
```

Listing 3.1: API Functions implemented on a per-platform basis

At a cursory glance, to support a platform, the platform must provide functionality for three tasks: start a global time for the program, have an API to retrieve the current global time, and wait for time to advance to a certain point. Arduino programs automatically have a hardware timer started on power up and also have an API to retrieve the current hardware time with microsecond granularity. To sleep, we busy wait in a loop by retrieving the hardware time repeatedly until we hit a certain time to advance past the loop. To support physical actions (i.e., an external event such as a button press occurs which should interrupt the sleep), we allow for breaking out of such a loop if an asynchronous event occurs.

Tasks like retrieving hardware time require entering and exiting critical sections to prevent race conditions. On unthreaded Arduino programs, we initiate a critical section by disabling all interrupts and re-enabling them upon leaving a critical section.

Threaded Runtime

The threaded runtime for Mbed-RTOS Arduino Boards has similar functionality to the unthreaded runtime for starting global time, retrieving global time, and sleeping on a per-thread basis. However, the concept of critical sections is different as we now have multiple threads rather than a single thread attempting to access code sections. Though we would ideally want to stick to an unthreaded runtime given LF's comparable performance to an RTOS [17], federated execution of Lingua Franca relies on Reactor-C's threaded runtime. To allow for future federated deployment onto compatible Arduino boards, Arduino programs on MBED-RTOS use Reactor-C's threaded runtime that requires platform-specific implementation of Threads, Mutexes, and Condition Variables.

Mbed-RTOS Support on Reactor-C

Mbed-RTOS [19] is a C++ Constrained Embedded Real-Time Operating System that allows for basic OS functionality in low-overhead embedded platforms like Arduino. Boards like Nano 33 BLE and its derivatives allow for the integration of Arduino code with MBED code to provide the simplicity of Arduino with the multithreaded flexibility of Mbed.

Integrating C and C++ Code

Arduino is built on the concept of C-style APIs in a C++ compiler environment. We use Reactor-C as a library framework over Lingua Franca's C++ Runtime due to the efficiency of Reactor-C's queueing system alongside an emphasis on supporting constrained embedded platforms. This decision comes at the cost of creating C programs for C++ environments like Arduino's MBED-OS which by default have no compatibility without conversions between the two coding languages.

To account for this discrepancy, we create C-style APIs that Reactor-C uses to interface with CPP code that manages Threads, Locks, and Condition Variables, allowing us to use MBED's C++ APIs within Reactor-C without needing to convert Arduino MBED programs to use the Lingua Franca C++ runtime.

3.5 Results

Test Suite

For Lingua Franca's Regression Tests, we introduce a series of compile-only tests for each of the big five core platforms (AVR, megaAVR, SAM, SAMD, MBED) that tests whether a binary file can be made for a specified LF-Arduino program that is flashable. In theory, we could have our test suite run through a simulator, but there is not a viable program to simulate Arduino programs without physical hardware, so we utilize compile-only tests.

Compile-Only Test Results

Under our compile-only testing framework, we bundle our Reactor-C Arduino platform library and LF-generated code together and attempt to produce a binary file given a board core and board type. For each test, we create a minimal Arduino program in LF that Blinks on and off every half-second as seen here, replacing CORE and TYPE as needed for each of the tests:

```
/**
 * This example demonstrates a very simple blink program that will turn on and off an
 * LED on the Arduino Board with a 50% duty cycle switching every half−second.
 */
target C {
```

```
    platform: {
        name: "arduino",
        board: "arduino:CORE:TYPE"
    }
}
main reactor Blink {
    timer t1(0, 1 sec)
    timer t2(500 msec, 1 sec)
    reaction(startup) {= pinMode(LED_BUILTIN, OUTPUT); =}
    reaction(t1) {= digitalWrite(LED_BUILTIN, HIGH); =}
    reaction(t2) {= digitalWrite(LED_BUILTIN, LOW); =}
}
```

AVR

Under our compilation test for the AVR Core, we chose to test the most common Arduino board types in circulation in the Uno and the Nano.

```
arduino-cli Version: 0.32.2 Commit: 2661f5d9
Sketch uses 23616 bytes (76%) of program storage space. Maximum is 30720 bytes.
Global variables use 2503 bytes (122%) of dynamic memory, leaving -455 bytes for local
    variables. Maximum is 2048 bytes.

Used platform Version
arduino:avr 1.8.6
Not enough memory; see https://support.arduino.cc/hc/en-us/articles/360013825179 for
    tips on reducing your footprint.
Error during build: data section exceeds available space in board
lfc: fatal error: Error running generator
java.io.IOException: arduino-cli failure
```

Under our minimal LF test, we see that the Arduino-CLI fails to create the binary file due to the constraint on SRAM (dynamic memory). This indicates that Reactor-C is too large in conjunction with Arduino overhead to successfully compile Arduino boards for the common AVR boards. Such a result should not be unexpected: AVR boards are typically designed for smaller projects and for minor interactions with sensors and actuators. That said, the inability to flash on a common board does indicate a need to re-examine memory allocation in Reactor-C and better memory management techniques as a part of future work.

MegaAVR, SAM, and SAMD

MegaAVR, SAM, and SAMD in general are much larger boards memory-wise than AVR and are designed for larger projects in general (more pin-outs and larger microcontrollers).

Without loss of generality between boards, we opt to test the SAM family using an Arduino Due board under the SAM core.

```
arduino—cli Version: 0.32.2 Commit: 2661f5d9
Sketch uses 61780 bytes (11%) of program storage space. Maximum is 524288 bytes.

Used platform Version
arduino:sam 1.6.12
SUCCESS: Compiling generated code for BlinkSAM finished with no errors.
********
Code generation finished.
```

Despite the greater sketch overhead due to the core's larger size, we have ample program storage space, and the compiler successfully compiles the binary file. When testing on a Due board, the program as expected blinks every half-second, confirming the minimal example.

MBED

Compared to the SAM/SAM-like family of boards above, MBED uses the threaded implementation of Reactor-C instead of the unthreaded implementation. As such, this tests whether the threaded implementation of Reactor-C in conjunction with C-style API calls to MBED APIs and the actual MBED library can all be flashed onto a single board.

```
    arduino—cli Version: 0.32.2 Commit: 2661f5d9
Sketch uses 123876 bytes (12%) of program storage space. Maximum is 983040 bytes.
Global variables use 44352 bytes (16%) of dynamic memory, leaving 217792 bytes for local
     variables. Maximum is 262144 bytes.

Used platform Version
arduino:mbed 3.3.0
SUCCESS: Compiling generated code for BlinkMBED finished with no errors.
********
Code generation finished.
```

As indicated by the test, there is ample room for threaded Reactor-C onto the Arduino MBED core. Additionally, the minimal Blink test works as expected, with threads workers correctly blinking the LED of the Arduino on and off every half second.

For future works, we would benchmark the real-time performance of LF-Arduino programs and their Arduino-only analogs, but we focus on the memory overhead aspect for this report.

Chapter 4

Compiling and Running Federated Programs

4.1 From Federation to Federates

As described in Chapters 1 and 2, we want to have the ability to execute Lingua Franca programs in a distributed fashion while still taking advantage of LF's guarantee of determinism. In particular, we want to treat federates as separate programs so that they can be independently deployed across nodes in a network. An ordinary LF program is turned into a federated one simply by changing the "main" keyword to "federated." Under the hood, this leads to a completely different compilation flow, the stages of which can be described at a high level as follows:

1. For each connection between two reactor instances in the (top-level) federated reactor, replace the connection between them with network proxy reactions. These proxy reactions are comprised of network output reactions for the downstream federate and network input reactions for the upstream federate.

2. For each reactor instance in the (top-level) federated reactor, generate an LF program that represents a federate. This program includes the corresponding reactor instance along with the inserted proxy reactions to let the reactor communicate with its up and downstream neighbors that will reside in other federates.

3. Compile each generated LF program separately.

The special handling of connections to enable networked communication and mapping specific parts of the program to individual federates is carried in a separate part of the compiler. Having this staged approach for the compilation of federated programs allows us to reuse existing code generator implementations without alteration. Separating the federation-specific compilation logic from the target code generation backend is important from a standpoint of modularity and code maintainability. Though we ultimately want

to treat each federate like a standalone LF program, there are subtle differences between an ordinary unfederated LF program and one that is part of a federation. As it turns out, some critical information about this bigger context of the federation needs to remain accessible to generate correct code for individual federates. As we will see, this necessary information is erroneously omitted from the individual federates, which can lead to incorrect level assignments.

Cyclic Dependencies between Federates

As discussed in the Composition subsection of Chapter 1, we need to reject programs with causality loops because such programs are not schedulable. In federated programs, where the coordination problem is split between independent runtime environments with their own event queue and reaction queue, the presence of cyclic dependencies between federates can pose a problem even if the cycle does not constitute a causality loop.

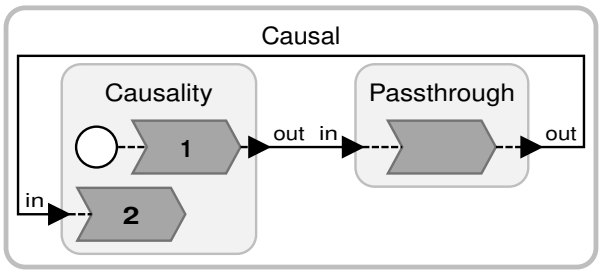

Figure 4.1: A reaction graph that does not have a causality loop even though a reactor has zero-delay feed through from an output port back to its input port.

In Figure 4.1, we have an unfederated program that is schedulable. There is no causality loop. Reaction #2 of Causality depends on the reaction in Passthrough, which depends on reaction #1 of Causality, which is triggered by the startup trigger. As such, the program can successfully compile; we can assign levels to each of the reactions.

Now let us define the Causal program to be federated instead. The federate generator program of the Lingua Franca compiler will see Causality and Passthrough as separate reactor instances. As such, it will replace the connections between each of them with network proxies and generate separate LF programs for each of them. We can examine the generated Causality federate more closely in Figure 4.2.

The causality federate's bottom reaction chain is responsible for receiving network inputs from passthrough while the top reaction chain is responsible for sending network outputs to passthrough. By going from federation to federate, we lost information about the Causality reactor that is crucial, namely that the top reaction chain must precede the bottom chain to maintain the precedence relation of the federation as a whole. As such, a naive level assignment algorithm may assign levels as indicated with the red numerals in Figure 4.3.

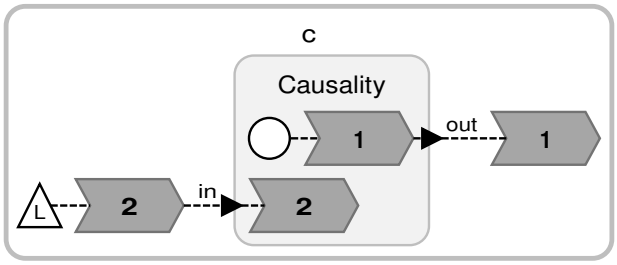

Figure 4.2: The generated Causal federate program as a result of treating the original LF program as federated.

As we shall see, an assignment like this will become a problem when we allow zero-delay feedback through other federates, *even though this does not constitute a causality loop.*

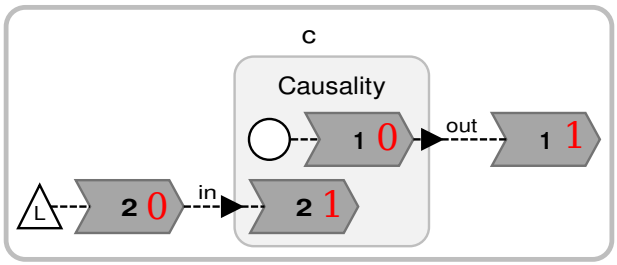

Figure 4.3: The erroneous level assignments for the Causality federate.

In our example, under a level-based execution strategy, the network output reaction has a spurious dependency on the network input reaction for the federate since the network input has a lower level than the network output. As such, the program deadlocks or halts since the Causality federate can never send a message to the Passthrough federate needed to allow the input reaction to proceed. Though there is no causality loop in the original LF program and in the generated federate, spurious dependencies in the level assignment algorithm on the federate cause the program to halt as if there was a causality loop. In the TypeScript target, cycles in federates are explicitly disallowed due to the complexities of handling this corner case. The C target does attempt to tackle this problem through special "unordered" reactions that are allowed to race to ensure progress.

4.2 Current Implementation in Reactor-C

The current implementation of Reactor-C attempts to preserve determinism while accounting for the spurious dependency problem between network output and network input reactions through the use of unordered/control reactions and phantom dependencies.

Unordered Reactions

Network Reactions (Input, Input Control, Output, and Output Control) are labeled at compile time and recognized at runtime as "unordered." Ordinary reactions enjoy mutually exclusive access to the state variables of the reactor through a static ordering between the reactor's reactions based on the lexical ordering of their definitions in the code. Unordered reactions assume no such ordering with respect to other reactions within the same reactor. While this is unsafe in general, it is safe for network reactions because we know that their code-generated reaction bodies do not access shared resources. It is important to note that an unordered reaction is not unordered with respect to reactions that supply it with inputs or use its outputs. These dependencies remain implied by its triggers, sources, and effects.

In Lingua Franca's federated C runtime, every port (input and output) has an associated network reaction that is placed at the top-level reactor. In instances where we have zero-delay feed-through between federates that produce a reactor cycle (as is the case in the Causal example), we add an additional set of unordered reactions called "input control reactions" whose primary purpose is to block progress of a reaction chain until we know the status of an input network port (absent or present with a value). Their dual, dubbed "output control reactions" are used to notify downstream federates when no output has been produced at the current tag (i.e., it is absent).

At execution time, the input control reactions are automatically enqueued at the start of the time step and will block progress until its associated network port status is no longer unknown. Once its status is known (absent or present), the network input reaction corresponding to the network port is enqueued and will send the network message to the federate where it will be processed. Once the entire reaction queue is emptied, if there are output ports that did not send messages, the network output control reactions associated with those output ports send an absent message to their destination federate if the value of the variable it tried to send was absent.

4.3 Benefits of Current Implementation

Phantom Dependencies

In the Causal example above, we need a way to nudge the level assignment algorithm for the individual federate to not give the network input reaction a lower level than the network output reaction to prevent deadlock. In an unordered reaction environment where all network reactions are on the top-level reactor, we can encode this requirement through a phantom dependency since all output ports are within scope.

During the federate generation process, when we create the associated network input reaction in the federate that is on the receiving end of a connection, we run a graph search algorithm from that receiving federate's input port. From said input port, we travel along upstream zero-delay connections, ports, and reactions, and check whether we reach an output port that belongs to the federate where the search started. When this occurs, we know that

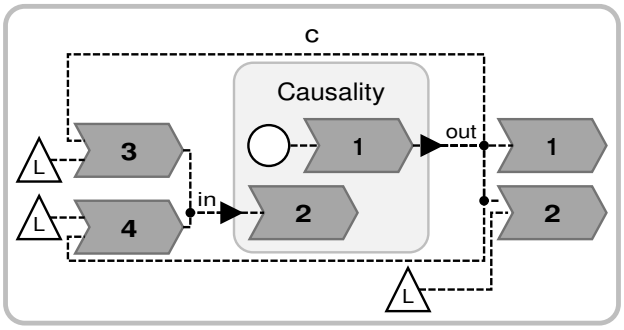

Figure 4.4: Inclusion of control reactions and a set of phantom dependencies gives us the necessary constraints from the context of the entire federation.

there exists zero-delay feedback between the specific output port and the input port (i.e., the input port network reaction is dependent on an output port network reaction of the same federate). Since the associated network reactions for these ports exist at the top level of the federate, we simply let each network input reaction declare a dependency on any output port that it is exposed to via zero-delay feedback. This nudges the level assignment algorithm to give the network input reactions at least the level as the network output reactions that are to supply them with inputs.

In Figure 4.4, the output of the Causality reactor gets fed back directly to both the network input and network input control reactions. Even though the network input reactions do not read the value of out, listing the output port of Causality among the sources they might read from induces a phantom dependency between the output reaction of Causality and its input reactions. This forces all unordered reactions to be on the same level in the reaction queue. When this occurs, the network input reactions are no longer forced to precede the network output reactions. Instead, there exists a race between worker threads to process these network reactions simultaneously. With careful design, the bodies of each of these reactions allow the reactions to occur in any order without violating determinism. As such, with enough threads, cycles that exist in the federate will no longer deadlock.

By keeping reactions at the top level (i.e., input and output connections of the network reactions co-exist in the main reactor), we can easily add the necessary connections to enforce the causal dependency.

Correctness and Simplicity of Network Input Control Reactions

Examining unordered reactions more closely, network input control reactions are crucial to preventing reactions downstream of network inputs from executing unless we know the status of a network input port. By definition of the reaction queue, reactions of higher levels are not allowed to execute at a certain logical time step until lower-level reactions have already

been executed. In the instance of the network input control reaction, its job is to simply idle at the same level as its associated network input reaction until it knows the port status.

In a centralized coordination context, this system forces downstream reactions of an input port reaction to correctly wait for a message from the RTI regarding said input port status. In decentralized coordination, each input port has a different Safe-To-Assume-Absent (STAA) time calculated at compile time that notates how long we should wait for a message on that particular port's status when we start a logical time step [14]. These input control reactions can be made to wait either for a signal that the port status has updated or for the STAA time. In the case that it reaches the full STAA time, we can safely assume we will not receive a message on that port for the current logical time step. Thus, the control reaction should update the port status to absent directly so that the network input can safely be unblocked. This will also allow any reaction downstream in the causal chain to proceed since the level is allowed to advance with the completion of both the input and input control reactions.

4.4 Drawbacks of Current Implementation

Thread Idling and Scalability

Though this approach ensures that port statuses are known before proceeding with executing downstream reactions, unordered control reactions introduce a thread idling problem where worker threads who are responsible for processing the reaction are forced to idle waiting for a port status to become known. In certain cases where we receive messages from multiple federates with input control reactions, there may not be enough worker threads to be able to handle all inputs. LF calculates in advance the maximal number of threads it needs for the particular program and will refuse to start executing the program if it lacks a sufficient number of threads. Under such a scheme, we require the number of worker threads to scale with the number of network inputs, which may not be practical on low-overhead embedded devices that lack the hardware to spawn a large number of threads. Additionally, having a lot of network reactions at the top level makes can make it difficult to grasp the meaning of the program. Specifically, if federates have multiple upstream and downstream neighbors, debugging generated target code (or the compiler that generated it) may become tedious due to an abundance of generated reactions at the top level that are difficult to trace back to connections in the federated LF program.

Symmetric Federate Deadlocking

Though phantom dependencies handle intra-federate causal dependencies, there are situations where level-based execution can still lead to deadlock due to the internal structure of the federate.

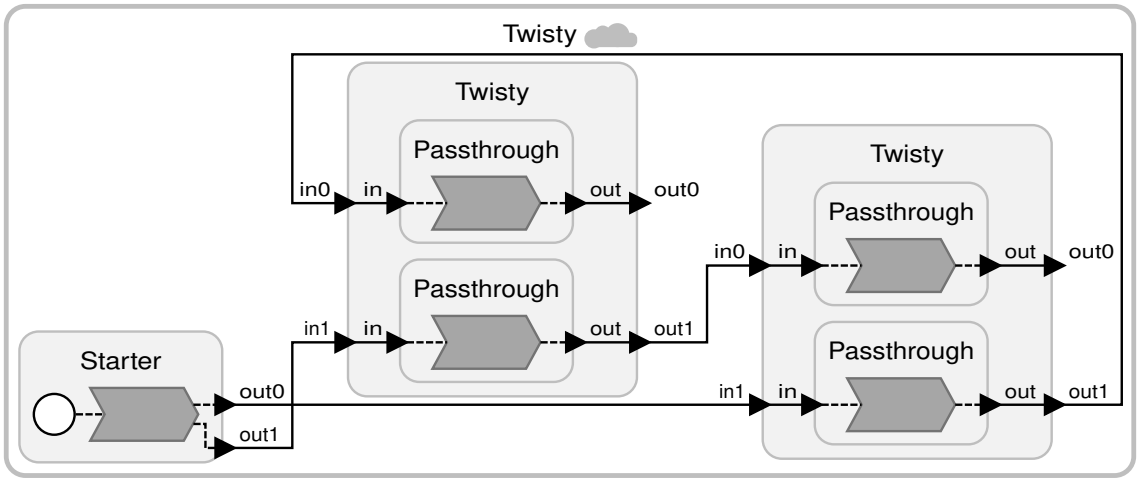

Figure 4.5: An example of a federation that still has deadlock on a per-federate level based on symmetry of the twisty reactors.

Consider the example in Figure 4.5 (developed jointly with Peter Donovan) in which we have input ports being routed to an associated output port through a passthrough reaction and output ports being routed to the opposing input port in the opposite federate. All input ports for Twisty come from another federate, so every input port will have an associated network input reaction. As such, we will get two Twisty federates with the following structure shown in Figure 4.6.

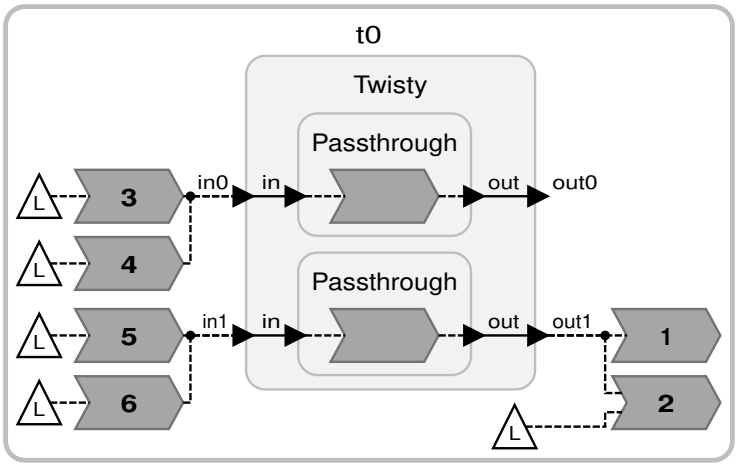

Figure 4.6: The generated structure of the Twisty federate that results in deadlock even with encoding intra-federate dependencies and deprioritization.

When we attempt to run a level assignment on such a structure, for each federate individually, we will produce the "valid" level assignment illustrated in Figure 4.7. But, as we shall see, this assignment will lead to deadlock.

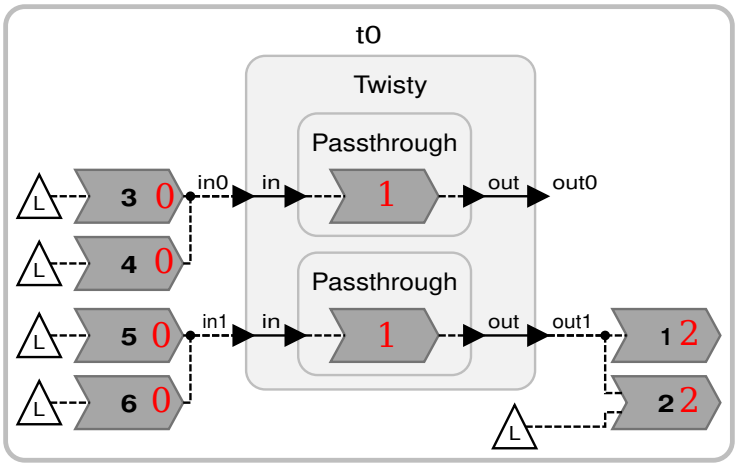

Figure 4.7: A possible level assignment of the Twisty federate that results in deadlock even with encoding intra-federate dependencies and deprioritization.

The network input control reactions are all at level 0, meaning they will execute first and block until their network port becomes known, preventing the network output reactions at level 2 from executing. This is due to symmetry since the top network receiver reaction gets a message from the opposing federate's network output reaction. Assuming both federates get equivalent level assignments, the federate will halt waiting for the top network receiver to get a message since we can never go past level 0, leading to deadlock.

However, there *does* exist a proper assignment that correctly deprioritizes the top network input control reaction to allow the network output reaction to proceed (see Figure 4.8), thus avoiding the deadlock, but we need an additional set of information to know which set of network receiver reactions to actually deprioritize.

The vulnerability to deadlock that the current level assignment scheme has is clearly problematic. Considering that the existing solution for handling circular dependencies between federates may pose a challenge to resource-constrained systems with their demands on thread creation, we look for a solution that addresses both problems.

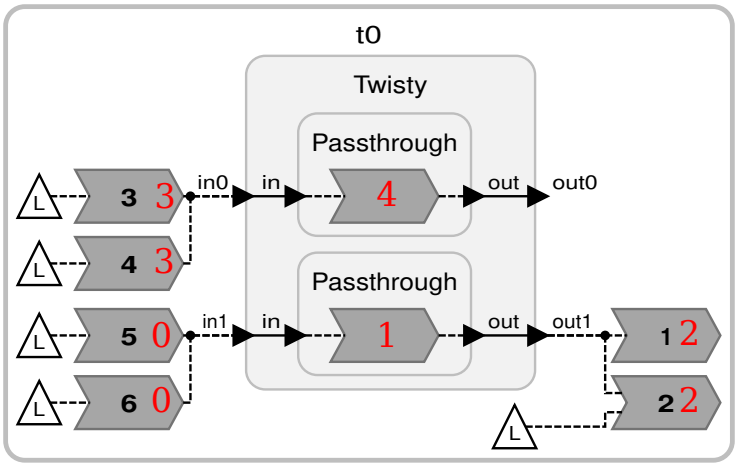

Figure 4.8: A correct level assignment of the Twisty federate that avoids deadlock, but requires information from the federation as a whole.

Chapter 5

Correct Handling of Circular Dependencies Between Federates

As was described in Chapter 4, the primary complications in the C implementation of federation stem from the following problems:

1. Unordered reactions force worker threads to idle until port statuses are known. These reactions require special treatment and the scheme requires a minimum number of worker threads to ensure progress. Constrained embedded systems are limited in the number of worker threads they can spawn, so programs may fail to run due to them requiring more threads than can be allocated.

2. Knowledge from the entire federation erroneously gets abstracted away when splitting the program into individual federate programs. The level assignment algorithm creates spurious dependencies that can lead to deadlock if the valid level assignment does not account for context from the entire federation.

5.1 Simplified and Modular Compilation

Rather than avoiding deadlock in the federated execution by resorting to unordered reactions that manage the exchange of messages between federates, we should be able to find a partial ordering of reactions in each federate (including the ones that relay messages) that achieves this.

One approach would be to create separate reactors for each network port to contain our network input and network output reactions. We can see a visualization of this in Figure 5.3. Inside our network input reactors on the left, we have a logical action and reactor. When we receive a message from an upstream federate, the runtime will automatically trigger the action associated with that upstream federate's id. When the action is triggered, it will enqueue the reaction that shuttles the message to one of the input ports of the federate.

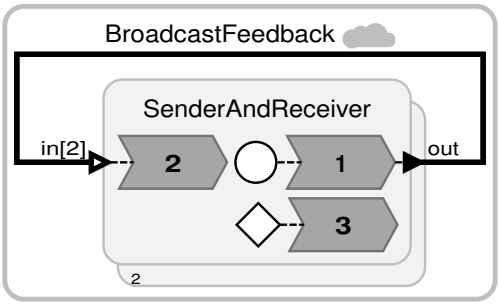

Figure 5.1: Example of a federation that showcases the differences in generation of federates when such federates require multiple input and output proxy reactions.

Our network output reactors on the right work in reverse. It will retrieve any outbound network messages and send them over the network to its corresponding downstream federate. These network output reactors also can come bundled with a logical action-triggered reaction (when this connection is a part of a federate cycle) that will send absent messages if the reactions do not produce an output by the current logical time step. Such a reaction is identical to the previous implementation with the only change being that the reaction is contained inside of our network output reactor rather than the top level.

This solution accounts for the message transfer in a way that does not require the use of any "unordered" annotations. At the same time, we no longer require a check on the minimal thread count since the number of threads we need no longer scales with the number of upstream federates. Now, we simply have existing worker threads process input reactions as messages over the network come in. We no longer require worker threads to idle waiting for port statuses as we did with input control reactions. However, we still require a mechanism to replace the network input control reactions that ensure that downstream reactions are barred from executing until their upstream network input port statuses are known. Rather than having a separate control reaction for each network input port, we can fulfill this constraint directly in the runtime by controlling dynamically the level up to which reactions are allowed to execute.

5.2 Max Level Allowed to Advance Condition (MLAA)

Under our system of level assignments, we require a mechanism for preventing downstream reactions that are causally dependent on network inputs from proceeding until we know the status of that network port. Rather than have a separate reaction to track each port, we can instead suspend the handling of reactions in the reaction queue based on what level we can safely advance to.

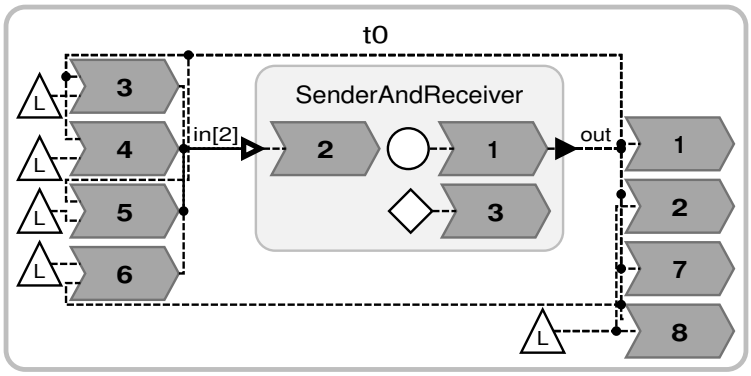

Figure 5.2: Diagram showcasing the old style of generating federates from the federation in Figure 5.1. This example shows inbound connections from two separate federates and outbound connections to two separate federates with all network reactions at the top level.

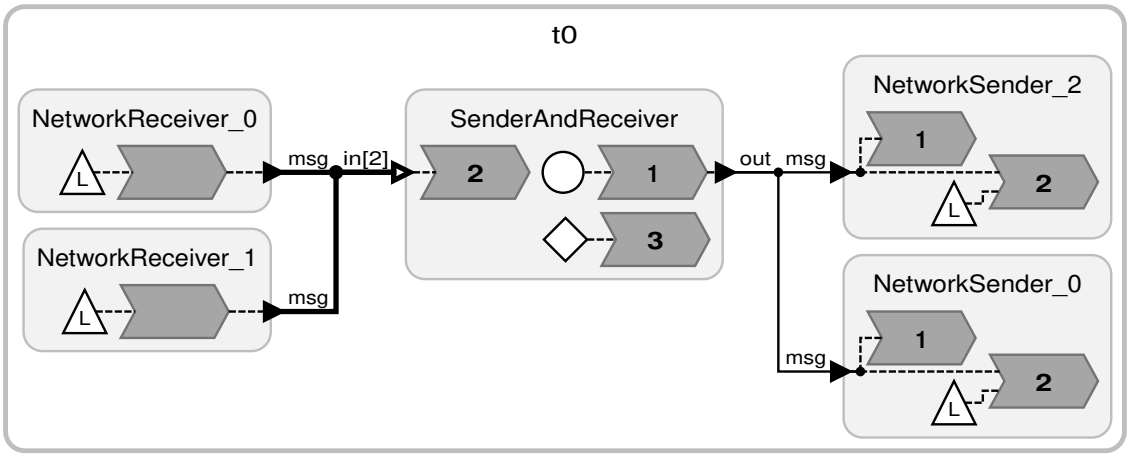

Figure 5.3: Diagram showcasing the new modular style of generating federates from the federation in Figure 5.1. Rather than having separate reactions on the top level of the federate, for every connection between federates, create a network sender reactor in the source federate and a network receiver reactor in the destination federate. This example shows inbound connections from two separate federates and outbound connections to two separate federates.

Let N be the set of all network input reactions for a federate. Each of these reactions will automatically be assigned a hard-coded level when an LF program is generated. We can further define two sets N_k, $N_u \subseteq N$ for reactions whose network port statuses are known and unknown, respectively. Also, let M be a variable representing the max level of reactions the runtime is allowed to execute based on the contents of N_k and N_u.

At runtime, when we try to process all reactions in the reaction queue at a logical time step, we go in ascending order starting from level 0. Once we try to advance to a level $k > M$, we suspend the processing of reactions until $M \geq k$.

At the start of the logical time step, $N_u = N$ since we do not know the network status for any of the ports. As such, we set M to the minimum value of reaction level across $\forall n \in N_u$.

Suppose we receive a network message that makes a certain input port status known (call its associated reaction N_0). We now let $N_u = N_u \setminus N_0$ and $N_k = N_k \cup N_0$. We again set M to the minimum value of reaction level across $\forall n \in N_u$ and signal for the reaction queue thread worker to check if it can safely advance again.

Once $N_u = \emptyset$, we can finally set M to the reaction with the max level of the entire federate since we can safely execute the rest of the reactions on the queue and notify the worker to proceed.

Centralized Coordination

Under a centralized form of coordination, before executing a downstream reaction, we need to make sure we know the status of every upstream network port to said reaction. When we receive a message from the RTI that changes the status of a particular port, we update the max level based on this information and send a signal to the reaction queue thread to check if it is safe to advance to subsequent levels. If the max level is at least equal to this downstream reaction's level, then the downstream reaction is allowed to execute.

Decentralized Coordination

Under a decentralized form of coordination, messages between federates do not flow through the RTI but are exchanged between federates directly. A federate no longer receives a notification from the RTI to signal that a port will be absent at the logical time step; instead, federates observe a wait time before they conclude that it is safe to assume the value of a port will be absent. In other words, given a logical time step L, after observing a Safe-To-Assume-Absent (STAA) offset, the federate assumes that it will not receive any message from upstream federates that would amount to the port being present at L. If this happens nonetheless (after the wait was observed), this means the deterministic reactor semantics were compromised, which implies a fault condition.

The STAA requires the runtime to either receive a message or wait until the physical time reaches $t = L + STAA$ before it can conclude what is the status of a port at L. When it reaches this physical time and it has received no message from the upstream federate for L, it can safely assume the value for the port will be absent. When this occurs, the runtime

is to set the port status for this associated reaction to absent and update the max level accordingly to advance the max level we are allowed to execute.

For the C runtime, we orchestrate this checking by spawning one additional thread on federate startup. This thread is responsible for managing a static list of STAA structs that contain an STAA offset and the list of network ports associated with this STAA offset. At the start of a logical time step, the thread will iterate through the list. For each STAA value, the thread will idle wait until it hits the offset, then set port statuses for the associated port reactions accordingly, and continue with the next STAA. Once it finishes every STAA value, the STAA thread will wait for the next logical time step before restarting and going through the list again. We also add a mechanism to signal the STAA thread to early restart if the next logical time step occurs before the thread has reached the end of the list.

With this approach, however, we still need to take care to assign levels such that any spurious dependencies implied by the levels do not cause deadlock.

5.3 Necessary Constraints on Network Reactions

Deprioritization of Select Network Input Reactions

Under our system of level assignments, we have an infinite number of possibilities for how to assign levels to each reaction. Under our new system of modular network input and output reactions, we could still have the issue of reactions R within a federate having spurious dependencies on network input reactions N that are not in the network input reaction's downstream causal chain. Under this system, reactions in R are not allowed to execute until all network input reactions N are executed, which is inefficient. At the same time, a valid level assignment can lead to deadlock in circumstances where an output reaction in a federate must precede an input reaction in the same federate without direct feedthrough.

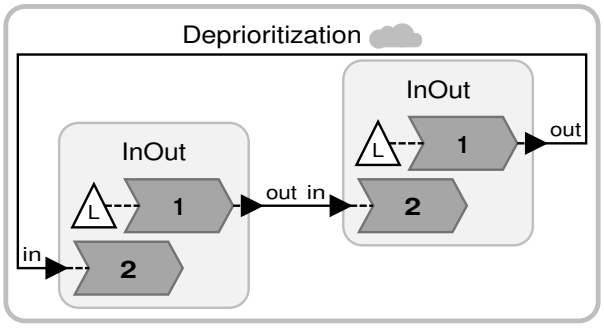

Figure 5.4: Diagram showing an example of a federation that requires deprioritization to prevent deadlock.

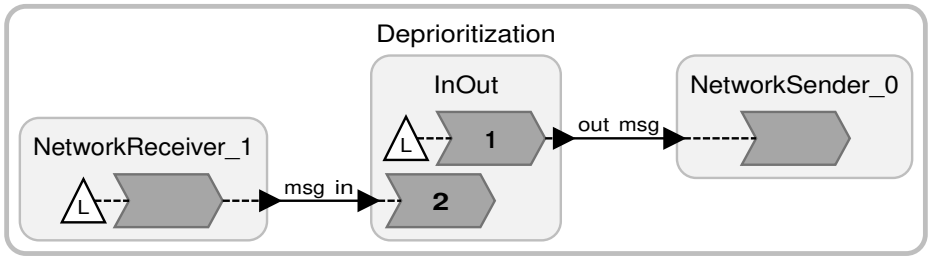

Figure 5.5: Diagram showing an example of a federate generated from Figure 5.4 that will without proper care produce a level assignment that can lead to deadlock.

In Figure 5.4, based on the global context of the federation, we know that reactions labeled 1 must be allowed to execute before reactions labeled 2 of each InOut federate to correctly prevent deadlock.

When the program for each InOut federate is generated as shown in Figure 5.5, we need to prioritize the top chain to precede the bottom chain (i.e. have network output reaction chains with no causal dependence on network input reactions to go as early as possible).

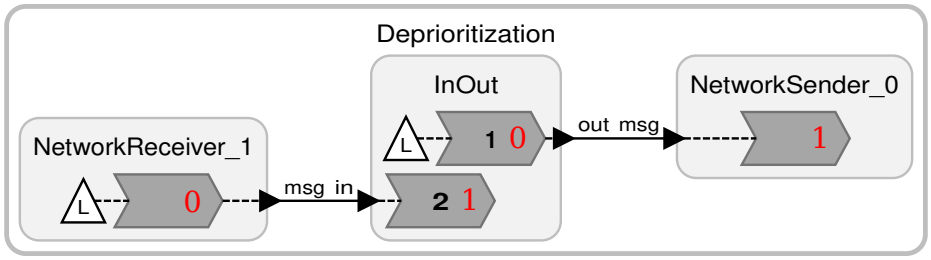

Figure 5.6: A valid level assignment for the "deprioritized" example that still causes deadlock. Under this system, we need the network sender from the opposing federate to send a message to the current federate before we can leave level 0, which will never happen since in both federates, the network sender is stuck waiting at level 1.

To resolve this, we need to deprioritize input reactions such that the level assignment algorithm holds off assigning levels for network input reactions. In practice, we should assign levels to reactions accordingly:

1. Assign levels for reactions within a federate and network output reactions that are not causally influenced by network input port reactions.
2. Assign levels for network input reactions.
3. Assign levels for causal downstream reactions from these network input reactions.

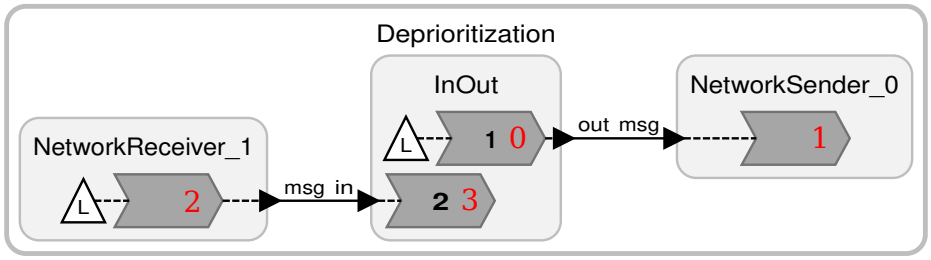

Figure 5.7: A valid level assignment for the deprioritization example that avoids deadlock by waiting to assign network receivers levels if network sender reactions are not in the same causal chain as said network receivers.

Deprioritization remedies the problem of certain reactions being blocked from executing due to spurious dependencies on network input reactions that exert no causal influence on them but simply happen to have been given a lower level in the level assignment algorithm. Though this creates spurious dependencies in the other direction (i.e., the network input reaction cannot proceed until the internal reactions go first), we are able to do as much work as possible within the federate, so threads will not have idle time until we need to wait for the network input. At the same time, as in the above example, deprioritizing the network input reactions can avoid certain assignments from deadlocking.

However, due to the level assignment occurring on a per-federate basis, even with deprioritization of the level assignment, the level assignment could cause deadlock. The reason for this is that there may be dependencies implied by the structure of the federation that are not captured in the federate program.

Encoding Intra-Federate Causal Dependencies (Federate Cycles)

When we convert from a federation to a set of federates, even though we replace connections between federates with associated network input and network output connections for each federate connection, we lose information that is needed to produce a level ordering assignment that avoids deadlocks.

Simply treating each federate in isolation poses the risk of ignoring the presence of zero-delay feedback that is established between its ports zero-delay paths through the rest of the federation. Looking at the Spiral example in Figure 5.8, ignoring such feedback would mean that the causal relationship between its first network output port's reaction and its second network input port reaction would not be recognized. To ensure such causal relationships are captured accurately, we need to nudge the level assignment algorithm to produce a proper level assignment given additional constraints.

Let P be the set of causal, zero-delay relationships between network output ports and network input ports for a particular federate. Prior to converting from federation to federates, we replace connections between federates with network proxy reactions (i.e., network input

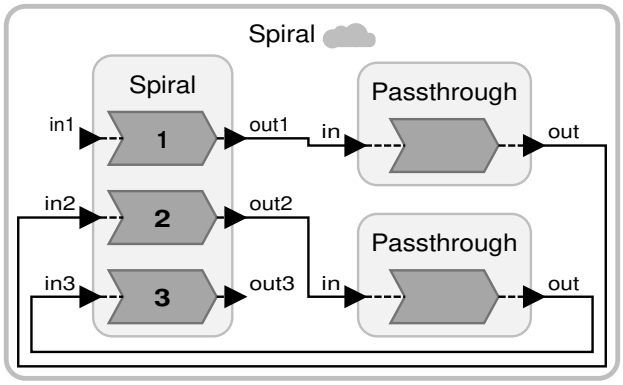

Figure 5.8: A more complex example of federate cycles called the Spiral problem. We require encoding causal dependencies between the reactions of the output reactors and the reactions of the input reactors.

reaction in the destination federate and network output reaction(s) in the source federate). For each input port P_i in the destination federate, we check if there is a zero-delay connection between this port and an output port P_O of the destination federate, and if so, add pair (P_O, P_i) to set P. When we finally produce the federate's LF file, we encode P's information into the file.

For the Spiral example, P =(out1,in2) $\bigcup$ (out2,in3) since out1 has zero-delay logical feed through to in2 and out2 has zero-delay logical feed through to in3.

During the level assignment stage, we use P to create an additional set of constraints on reaction ordering. Given this additional set of constraints, for output-input paired reactions R_o and R_i, we can guarantee that level(R_o) $<$ level(R_i). Under this system, we add enough constraints to correctly assign levels to the spiral problem since we encode the intrinsic dependency between the first output port and the second input port.

Symmetry

Looking back on the symmetry example of Twisty in Chapter 4 (Figure 4.5), we still need a way to encode enough information from the federation to break the symmetry and correctly prioritize the right network receiver reactions. Under our new framework (Figure 5.9), we still have the same symmetric interactions as we had before. As such, the level assignment will remain identical. There's no cyclic feedback to the same federate and all chains rely on network input reactions, so we do not have enough information to correctly prioritize the bottom chain of reactions over the top chain.

We still need to find a way to capture the dependencies outlined above while including additional federation-level dependencies to overcome deadlock induced by symmetric relationships between federates.

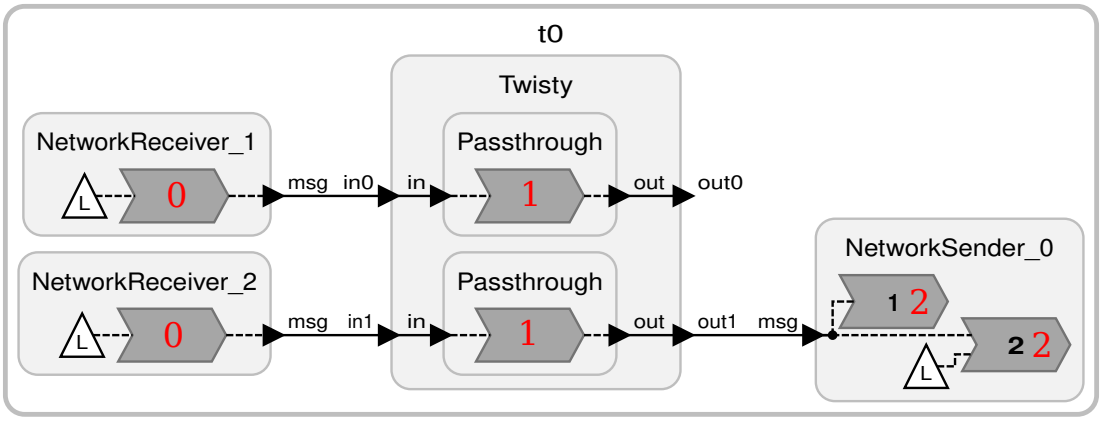

Figure 5.9: A level assignment on the generated federate of the Twisty Symmetric example under our new system of handling circular dependencies. We still lack the necessary information from the entire federation needed to correctly order reactions even with intra-federate encodings and deprioritization. As such, we still run into the same deadlock problem as the old implementation.

5.4 Total Port Ordering (TPO)

We conjecture that adhering to a total port ordering amongst upstream and downstream federate ports across the entire federation gives us a deprioritization strategy for achieving a deadlock-free level assignment on a per-federate basis. We will illustrate the approach based on the Twisty example described in Chapter 4 (Figure 4.5).

Overview

Total Port Ordering (TPO) is a strategy designed jointly with Peter Donovan and Marten Lohstroh to provide necessary information from the federation to individual federates by leveraging information provided through global-level assignments. The workflow of the process is as follows:

1. Do a federation-wide global assignment of levels to reactions and network ports (ports that connect one federate to another)

2. When converting from the federation to the federates, for every network proxy reactor generated in place of a network port, annotate said proxy reactor with the global level assigned to its network port from the federation.

3. When the reaction graph is generated for the federate prior to level assignment, add relative dependencies between network proxy reactions in ascending order based on the relative ordering of the annotated levels of each of the containing network reactors.

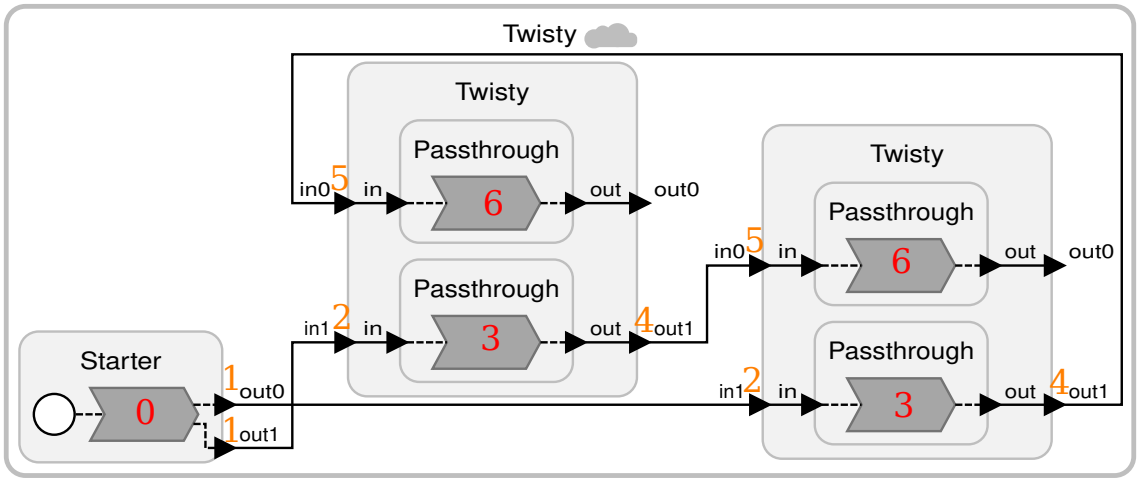

Figure 5.10: An example of total port ordering on an entire federation denoted in orange for the ports and red for the reactions. We assign Starter's output ports to level 1 since they have upstream dependencies on the starting reaction at level 0. The bottom input and output ports of Twisty have level values of 2 and 4 respectively and the top input port of Twisty has a value of 5. The value of out0's port level in the Twisty federates is omitted as it is not connected to anywhere else in the network.

When we apply our TPO strategy to the Twisty example, we get the following federate (precluding level assignments) in Figure 5.11.

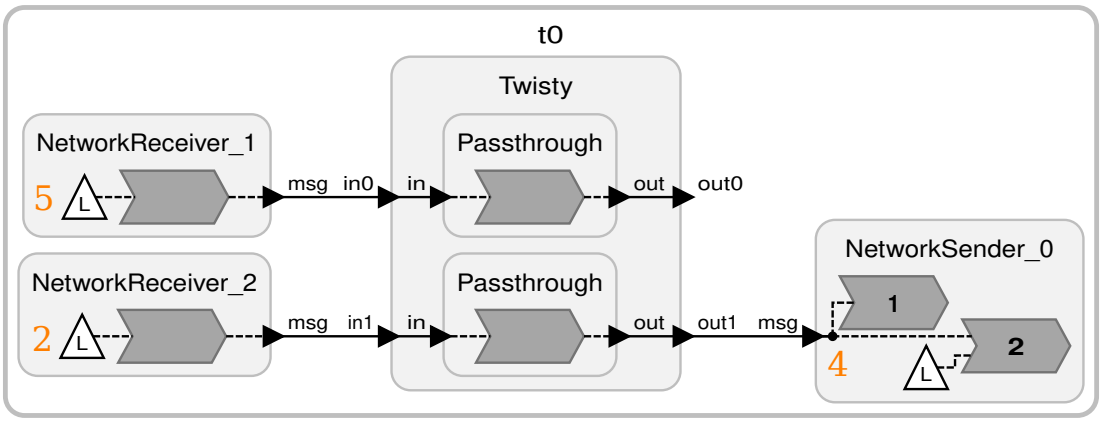

Figure 5.11: An example of total port ordering values being transferred to federates through the network proxy instantiation of said ports as indicated by the values in orange.

Given this set of information, we ignore the magnitude of the level assigned and instead focus on the relative ordering of the reactions based on the port levels. We then sort the

network reactions by these levels and create phantom dependencies between successive levels (i.e., have all port level $n+1$ reactions be dependent on the port level n reactions, skipping below level n as needed until we hit a port level for the federate that has reactions on said level). By the transitive property, these successive chains will ensure all port level n reactions be dependent on all port k reactions such that $k \in [0, n-1]$

In the Twisty example, NetworkReceiver_1's reaction will be given a phantom dependency on NetworkSender_0's reactions since NetworkReceiver_1's port level is higher than NetworkSender_0's port level. This will cause NetworkSender_0's reactions to execute before NetworkReceiver_1's reactions, which is exactly what is needed to avoid deadlocking. The resultant correct level assignment is shown in Figure 5.12.

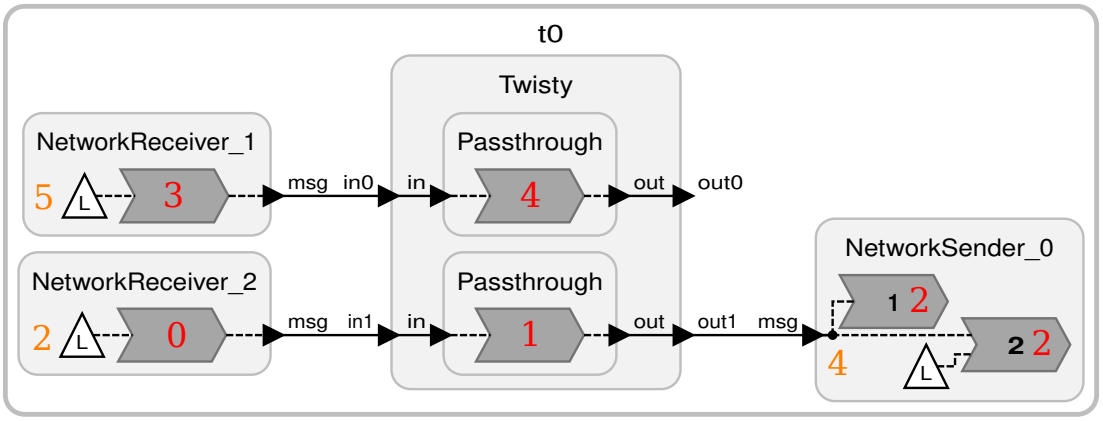

Figure 5.12: An example of a correct level assignment for the Twisty example with reaction levels denoted in red and inherited port level values denoted in orange. Providing the total port ordering to the federate and adding relative dependencies gives enough dependency information to correctly assign levels.

One design consideration to consider is the choice of only providing the generated proxy reactions their global-level assignment. Even though we do a global assignment of all network ports and reactions for the federation, we elect to only pass down the level assignment pertaining to the network ports and exclude passing down the levels assigned to reactions down to the generated federates.

For TPO, we only attempt to create phantom dependencies between network reactions to prevent deadlock due to network-wide dependencies that are lost when converting from federation to federates. Though the internal reactions of a federate affect how levels are assigned to the proxy reactions, our goal is to encode a partial ordering on the order of network reactions to force network reactions in a federate to go in the same order as it would have from a global context. As such, we only create dependencies between network proxy reactions based on the relative ordering of reactions from the global context.

TPO Constraints as a Superset of Federate Cycle Constraints

Theorem 1 *The TPO algorithm can provide the same constraints as when encoding intra-federate causal dependencies (federate cycles) on network sender and receiver reactions with zero-delay logical feed-through.*

Proof: Suppose P_o and P_i are output and input ports of the same federate that have a zero-delay logical connection between them through other federates on a network. Under TPO, suppose P_o has been assigned level n. When taking a walk through the federation from P_o to get to P_i, we must cross through at least one external federate which is guaranteed to increase the port level by at least 2 (i.e., a connection through a federate must cross an input port at least once and an output port at least once through said federate). As such, P_i is guaranteed to have a port level of at least level $n + 3$ since the port level goes up by two when crossing the external federate and up by one when assigning the network input port's level based on a connection from a network output port. By the transitive linking property of TPO, P_i's associated network reaction will have a causal dependency on P_o's associated network reaction due to having a higher port level. Thus, P_i's associated network input reaction will be forced to execute after P_o's associated network output reaction(s).

TPO Constraints as a Superset of Deprioritization Constraints

Theorem 2 *The TPO algorithm provides the same constraints as deprioritization with regards to allowing certain network output reactions to execute prior to certain network input reactions.*

Proof by cases:

- Case 1: A network output reaction has an upstream dependency to a set of network input reaction(s) of the same federate. Suppose the network input reaction with the highest port level assignment in the chain has level n, this network output reaction will be assigned at least level $n + 1$, implying that the set of all network input reaction(s) in the chain must precede the network output reaction. This behavior is expected since we need these network input reactions to be executed before the network output reaction. As such, we cannot deprioritize this network input reaction, so the expected behavior is maintained.

- Case 2: A network output reaction has no upstream dependency to network input reactions. Under our assignment system, such a reaction would be blocked from executing by a network reaction if it has a level greater than an input reaction belonging to the same federate. By our global-level assignment protocol, we can show that there cannot exist a causal relationship between the two reactions. Since we assume the network output reaction has no upstream dependency on a network input reaction, there cannot exist a connection between a network input reaction and a network output

reaction within the federate. On the other hand, we know there cannot exist a logical connection from the network output reaction to an associated network input reaction of the same federate by the logic outlined in the Federate Cycle Constraints section above. If there was a connection, then the network input reaction must have been assigned a level greater than the network output reaction, which is a contradiction. As such, network output reactions that have no upstream dependency have no causal dependence on other network input reactions, which will allow them to execute freely without being deadlocked.

Note: Under this level assignment system, certain network output reactions can get assigned port level values higher than network input reactions of the same federate even where there's no logical connection between said network output and network input reactions. Though this implies a spurious dependency, we tolerate this since reactions with a lower port level should execute before reactions with a higher port level from the perspective of the federation as a whole.

Causality Loop Prevention under TPO

The primary concern of TPO is adding too many constraints in reaction dependencies to the point where a cycle is produced in the reaction-directed acyclic graph. Causality loops will prevent the level assignment algorithm from being able to assign levels to the reactions.

Theorem 3 *The TPO algorithm does not introduce cycles into the underlying reaction graph.*

Proof by contradiction. Suppose we add an edge between a network reaction N_0 with level k and a network reaction N_1 with level l such that $l \geq k+1$. If such an edge introduces a causality loop, it implies there's a logical, zero-delay connection path between N_1 to N_0 such that N_0 is downstream of N_1. However, if N_0 is downstream of N_1 from the federation perspective, TPO would need to assign N_0 a higher port level than N_1 since levels strictly increase when hopping from one network port to the next. Thus, N_0 must have been assigned a level that is both strictly less than (needed to add an edge) and strictly greater than N_1's level, which is a contradiction since no overlapping level exists under these constraints. Thus, the addition of edges into the reaction graph based on Total Port Ordering cannot by itself introduce cycles.

Total Port Ordering constraints serve as a superset of deprioritization, encoding, and symmetric constraints. Though this set of constraints is not exhaustive to what may be needed, it serves as a stepping stone to formalizing proofs in future works and allows us to better understand conversions from federation to federates.

5.5 Comparing Current and New Implementations

Given our new modular framework and strategies of MLAA and TPO, we can compare the current implementation to our proposed implementation based on required threads and correctness when it comes to handling dependencies between network reactions.

Required Threads

When comparing the two implementations, we need to examine the minimum number of required threads needed for each coordination scheme. We first examine the number of threads required for the original implementation. Assume for our federate that we have p upstream federates and connections to k different federates in the network.

Current Implementation		
Coordination	Centralized	Decentralized
Program	1	1
Worker	$p+1$	$p+1$
Network	1	k
Total	$p+3$	$k+p+2$

Table 5.1: Table indicating the number of threads required to be spawned under the current implementation

Examining Table 5.1, since we need worker threads to idle on a particular input port status, we need at least p worker threads to handle all input port control reactions as well as one additional worker thread to process ready reactions while the other p worker threads are forced to idle. We also need at least one thread to handle the network connection to the RTI for centralized coordination and k threads to handle connections with k different federates for decentralized coordination. As such, we need $p+3$ threads at minimum for centralized coordination and $k+p+2$ threads for decentralized coordination.

In both decentralized and centralized coordination, the number of threads required scales with the number of upstream federates, which should be avoided on low-overhead embedded systems that may require multiple connections without the necessary overhead to spawn a scaling number of threads. As such, we require our proposed implementation to no longer have the scaling problem. We can examine the minimum number of threads for our proposed implementation:

Under our proposed implementation in Table 5.2, we require a minimum of only one worker thread since we no longer have network input control reactions that force threads to idle. As such, thread requirements no longer scale with upstream federates, and we only need 3 total threads (2 excluding the program thread) in centralized coordination. Three

Proposed Implementation			
Coordination	Centralized	Decentralized ($p = 0$)	Decentralized ($p > 0$)
Program	1	1	1
Worker	1	1	1
STAA	0	0	1
Network	1	k	k
Total	3	$k + 2$	$k + 3$

Table 5.2: Table indicating the number of threads required to be spawned under the proposed implementation

threads are trivial for many embedded systems with threading support, so we can very easily use centralized coordination on low-overhead embedded systems under the proposed implementation.

Decentralized coordination also no longer relies on p, and threads only scale with respect to the total number of connecting federates in the network. When there are no upstream federate ($p = 0$), we require the same number of threads in both the current and proposed implementation. When there is at least one upstream federate ($p > 0$), the number of threads required no longer scales with p since we only require a single thread that processes STAA port statuses.

Deadlocking

For a federate to successfully execute without deadlocking, we need a way to create phantom dependencies based on federate cycles, deprioritization of certain network input reactions, and constraints of the federation on a per-federate level. Under our current system of unordered reactions, we can successfully create phantom edges to avoid federate cycles from creating deadlock since every network reaction is at the top level. However, our current scheme fails to add dependencies in instances where we do not have federate cycles. This poses a major issue when we have cases like the symmetric Twisty example that require knowledge of the federation to know which chain of reactions to prioritize over the others.

Under our new strategy of TPO, we successfully can determine an ordering of different chains of reactions by encoding a relative ordering of network reactions based on the federation as a whole. As such, even in cases where we do not have federate cycles, we can still correctly order reactions so as to avoid deadlock.

Chapter 6

Conclusion

With regards to Arduino support, we see mixed success with regards to common boards of the AVR family as Uno and Nano boards lack the proper memory needed to store the vast size of the Reactor-C runtime. Conversely, boards like the Due, Zero, and Nano 33 BLE are flashable with the Reactor-C library and work as intended in both unthreaded and threaded environments.

On the note of the federated runtime, we can successfully generate our new modular framework for network inputs and outputs that preserve determinism through a proper flow between reactions without relying on unordered reactions. Similarly, such a modular framework no longer has a problem with requiring worker threads to scale with the number of upstream federates. Though MLAA and TPO implementations are in their infancy and require more extensive testing, they show promise in allowing us to maintain level assignment frameworks in Reactor-C without relying on the entire dependency graph at runtime so long as we tolerate spurious dependencies. As described below, the findings can be made more robust through future work.

6.1 Future Work

Optimizing Memory Footprint

Under the current Reactor-C implementation, we cannot successfully compile LF Programs on many common Arduino boards like the Uno and Nano since Reactor-C has too high of a memory footprint. One strategy is to use static memory in place of malloc calls. While malloc on Arduino boards is still valid, the use of dynamic memory repeatedly for space-constrained systems is not ideal. As such, Reactor-C should be optimized to include static memory for attributes and variables we know at compile time to optimize the memory footprint. Though there are Arduino-specific tricks to lower memory usage (placing constant string literals in Sketch memory, minimizing array sizes, and converting globally scoped variables to local when possible), a full optimization of Reactor-C would be crucial to ensure Reactor-C can

compile onto any Arduino board in the future.

Unthreaded Federation Support

Federation in Lingua Franca relies on the threaded runtime implementation since we need to monitor network messages alongside executing reactions in parallel. As indicated in the proposed implementation in Table 5.2, we require a minimum of 3 threads for centralized coordination and $k + 3$ threads in decentralized coordination if we have upstream federate connections (where k is the total number of federates we need to communicate with). We must consider how to compress the number of threads required in each of these schemes to only 1 when attempting to add unthreaded support for federated execution.

With centralized coordination, all communication is already serialized through the RTI for all connections between federates. To convert the work of the network and program threads to a single process thread, we would need to implement an event loop in Reactor-C. Event loops are used in many unthreaded embedded systems like Arduino where the single main process clears events one at a time in a specified order for the program life. For our federation, we populate a global event FIFO queue with events that need to be processed. When we receive a message over the network, it should trigger an interrupt or asynchronous request to add an event to the global queue to be processed later. We then clear the event loop over the life of the program. Such a queue would need to account for deadlines and high-priority events like shutdown and logical time advancement events and update the queue accordingly to prevent stale events.

Under a decentralized model, the unthreaded runtime would still need an event loop to process events. Compared to the centralized model, a decentralized model also requires a proper serialization of incoming messages from multiple federates concurrently since the current model relies on one thread per connection to federate neighbors. Additionally, there would need to be interrupt functionality to account for STAA checking of input ports in place of an extra thread since such events may become stale and delayed on the event loop. Certain architectures exist that could eliminate the need for blocking sockets by relaxing the synchronicity constraint between federates. One possible strategy is the modified form of a Loosely Time-Triggered Architecture [20]. Under such a scheme, we replace synchronous socket connections with asynchronous calls to Finite FIFO Platforms (FFPs) for each unidirectional connection between federates [21]. Such a solution is speculative and may require additional care to preserve LF semantics, but it is worth examining.

Networking on Arduino

Arduino as a microcontroller is designed primarily for serial communication along USB due to the overhead associated with having an onboard WiFi or Bluetooth Module. Certain Arduino boards like the Uno WiFi and Nano 33 BLE have support respectively for WiFi and Bluetooth but are either discontinued or cost prohibitive. There have been several IoT applications in healthcare [22], [23] where basic Arduino boards could connect over the

internet through the use of ESP-01 WiFi Modules connected directly to Arduino's Serial Pins to deliver real-time health data. These modules are high-integration wireless System-on-a-chip controllers with both a complete TCP/IP stack and standard IEEE802.1 b/g/n support. Adding support for networking using these modules in place of the standard POSIX sockets API would allow for federated support with any embedded device that utilizes Reactor-C for its framework.

Bibliography

[1] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. "Toward a Lingua Franca for Deterministic Concurrent Systems". In: *ACM Trans. Embed. Comput. Syst.* 20.4 (May 2021). ISSN: 1539-9087. DOI: 10.1145/3448128. URL: https://doi.org/10.1145/3448128.

[2] *Lingua Franca*. https://www.lf-lang.org/.

[3] Marten Lohstroh. "Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html.

[4] Marten Lohstroh, Íñigo Romeo, Andrés Goens, Patricia Derler, Jerónimo Castrillón, Edward Lee, and Alberto Sangiovanni-Vincentelli. *Reactors: A Deterministic Model for Composable Reactive Systems*. Jan. 2019.

[5] Edward A. Lee. "Determinism". In: *ACM Trans. Embed. Comput. Syst.* 20.5 (May 2021). ISSN: 1539-9087. DOI: 10.1145/3453652. URL: https://doi.org/10.1145/3453652.

[6] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A. Lee. *Xronos: Predictable Coordination for Safety-Critical Distributed Embedded Systems*. 2022. arXiv: 2207.09555 [cs.DC].

[7] Felice Balarin, Luciano Lavagno, Praveen Murthy, Alberto Sangiovanni-Vincentelli, C.D. Systems, and A. Sangiovanni. "Scheduling for Embedded Real-Time Systems." In: *Design & Test of Computers, IEEE* 15 (Feb. 1998), pp. 71–82. DOI: 10.1109/54.655185.

[8] Leonardo Mangeruca, Massimo Baleani, Alberto Ferrari, and Alberto Sangiovanni-Vincentelli. "Uniprocessor Scheduling under Precedence Constraints for Embedded Systems Design". In: *ACM Trans. Embed. Comput. Syst.* 7.1 (Dec. 2007). ISSN: 1539-9087. DOI: 10.1145/1324969.1324975. URL: https://doi.org/10.1145/1324969.1324975.

[9] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee. *High-Performance Deterministic Concurrency using Lingua Franca*. 2023. arXiv: 2301.02444 [cs.PL].

[10] T. C. Hu. "Parallel Sequencing and Assembly Line Problems". In: *Oper. Res.* 9.6 (Dec. 1961), pp. 841–848. ISSN: 0030-364X. DOI: 10.1287/opre.9.6.841. URL: https://doi.org/10.1287/opre.9.6.841.

[11] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. "The Department of Defense High Level Architecture". In: *Proceedings of the 29th Conference on Winter Simulation*. WSC '97. Atlanta, Georgia, USA: IEEE Computer Society, 1997, pp. 142–149. ISBN: 078034278X. DOI: 10.1145/268437.268465. URL: https://doi.org/10.1145/268437.268465.

[12] Yang Zhao, Jie Liu, and Edward A. Lee. "A Programming Model for Time-Synchronized Distributed Real-Time Systems". In: *13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07*. Apr. 2007, pp. 259–268. URL: http://chess.eecs.berkeley.edu/pubs/325.html.

[13] Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng, and Patricia Derler. "Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems". In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2009, pp. 77–86. DOI: 10.1109/RTAS.2009.39.

[14] Edward A. Lee, Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. *Consistency vs. Availability in Distributed Real-Time Systems*. 2023. arXiv: 2301.08906 [cs.DC].

[15] *Arduino*. https://www.arduino.cc/en/Guide/Introduction.

[16] Caleb Helbling and Samuel Z. Guyer. "Juniper: a functional reactive programming language for the Arduino". In: *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, Sept. 2016. DOI: 10.1145/2975980.2975982. URL: https://doi.org/10.1145%5C%2F2975980.2975982.

[17] Erling Rennemo Jellum, Shaokai Lin, Peter Donovan, Efsane Soyer, Fuzail Shakir, Torleiv Bryne, Milica Orlandic, Marten Lohstroh, and Edward A. Lee. "Beyond the Threaded Programming Model on Real-Time Operating Systems". In: *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*. Ed. by Federico Terraneo and Daniele Cattaneo. Vol. 108. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:13. ISBN: 978-3-95977-268-6. DOI: 10.4230/OASIcs.NG-RES.2023.3. URL: https://drops.dagstuhl.de/opus/volltexte/2023/17734.

[18] *Arduino CLI*. https://arduino.github.io/arduino-cli/.

[19] *Introduction to Arm Mbed OS*. https://os.mbed.com/docs/mbed-os.

[20] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. "A Protocol for Loosely Time-Triggered Architectures". In: *Proceedings of the Second International Conference on Embedded Software*. EMSOFT '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 252–265. ISBN: 354044307X.

[21] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto Sangiovanni-Vincent, Paul Caspi, and Marco Di Natale. "Implementing Synchronous Models on Loosely Time Triggered Architectures". In: *IEEE Transactions on Computers* 57.10 (2008), pp. 1300–1314. DOI: 10.1109/TC.2008.81.

[22] Md. Milon Islam, Sheikh Nooruddin, Fakhri Karray, and Ghulam Muhammad. "Internet of Things: Device Capabilities, Architectures, Protocols, and Smart Applications in Healthcare Domain". In: *IEEE Internet of Things Journal* 10.4 (Feb. 2023), pp. 3611–3641. DOI: 10.1109/jiot.2022.3228795. URL: https://doi.org/10.1109%5C%2Fjiot.2022.3228795.

[23] Anshuman Mishra and Richards Joe Stanislaus. *Smart Watch Supported System for Health Care Monitoring*. 2023. arXiv: 2304.07789 [eess.SY].