This copy of the Lingua Franca handbook was created on Thursday, February 10, 2022 against commit `8253e0`.

Table of Contents

Overview

Lingua Franca (LF) is a polyglot coordination language for concurrent and possibly time-sensitive applications ranging from low-level embedded code to distributed cloud and edge applications. An LF program specifies the interactions between components called reactors. The emphasis of the framework is on ensuring deterministic interaction with explicit management of timing. The logic of each reactor is written in one of a suite of target languages (currently C, C++, Python, and TypeScript) and can integrate legacy code in those languages. A code generator synthesizes one or more programs in the target language, which are then compiled using standard toolchains. If the application has exploitable parallelism, then it executes transparently on multiple cores without compromising determinacy. A distributed application translates into multiple programs and scripts to launch those programs on distributed machines. The communication fabric connecting components is synthesized as part of the programs.

Lingua Franca programs are compositions of reactors, whose functionality is decomposed into reactions, which are written in the target languages. Reactors are similar to actors, software components that send each other messages, but unlike classical actors, messages are timestamped, and concurrent composition of reactors is deterministic by default. When nondeterminstic interactions are tolerable or desired, they must be explicitly coded. LF itself is a polyglot composition language, not a complete programming language. LF describes the interfaces and composition of reactors. See our publications and presentations on reactors and Lingua Franca.

The language and compiler infrastructure is very much under development. An IDE based on Eclipse and Xtext is under development, and command-line tools are also provided. LF is, by design, extensible. To support a new target language, a code generator and a [[runtime]] system capable of coordinating the execution of a composition of reactors must be developed.

The C runtime consists of a few thousand lines of extensively commented code, occupies tens of kilobytes for a minimal application, and is extremely fast, making it suitable even for deeply embedded microcontroller platforms. It has been tested on Linux, Windows, and Mac platforms, as well as some bare-iron platforms. On POSIX-compliant platforms, it supports multithreaded execution, automatically exploiting multiple cores while preserving determinism. It includes features for real-time execution and is particularly well suited to take advantage of platforms with predictable execution times, such as PRET machines. A distributed execution mechanism is under development that takes advantage of clock synchronization when that is available to achieve truly distributed coordination while maintaining determinism.

Reactors

Reactors are informally described via the following principles:

1. *Components* — Reactors can have input ports, actions, and timers, all of which are triggers. They can also have output ports, local state, parameters, and an ordered list of reactions.
2. *Composition* — A reactor may contain other reactors and manage their connections. The connections define the flow of messages, and two reactors can be connected if they are contained by the same reactor or one is directly contained in the other (i.e., connections span at most one level of hierarchy). An output port may be connected to multiple input ports, but an input port can only be connected to a single output port.
3. *Events* — Messages sent from one reactor to another, and timer and action events each have a timestamp, a value on a logical time line. These are timestamped events that can trigger reactions. Each port, timer, and action can have at most one such event at any logical time. An event may carry a value that will be passed as an argument to triggered reactions.
4. *Reactions* — A reaction is a procedure in a target language that is invoked in response to a trigger event, and only in response to a trigger event. A reaction can read input ports, even those that do not trigger it, and can produce outputs, but it must declare all inputs that it may read and output ports to which it may write. All inputs that it reads and outputs that it produces bear the same timestamp as its triggering event. I.e., the reaction itself is logically instantaneous, so any output events it produces are logically simultaneous with the triggering event (the two events bear the same timestamp).
5. *Flow of Time* — Successive invocations of any single reaction occur at strictly increasing logical times. Any messages that are not read by a reaction triggered at the timestamp of the message are lost.
6. *Mutual Exclusion* — The execution of any two reactions of the same reactor are mutually exclusive (atomic with respect to one another). Moreover, any two reactions that are invoked at the same logical time are invoked in the order specified by the reactor definition. This avoids race conditions between reactions accessing the reactor state variables.
7. *Determinism* — A Lingua Franca program is deterministic unless the programmer explicit uses nondeterministic constructs. Given the same input data, a composition of reactors has exactly one correct behavior. This makes Lingua Franca programs *testable*.
8. *Concurrency* — Dependencies between reactions are explicitly declared in a Lingua Franca program, and reactions that are not dependent on one another can be executed in parallel on a multicore machine. If the target provides a distributed runtime, using Ptides for example, then execution can also be distributed across networks.

Time

Lingua Franca has a notion of logical time, where every message occurs at a logical instant and reactions to messages are logically instantaneous. At a logical time instant, each reactor input will either have a message (the input is **present**) or will not (the input is **absent**). Reactions belonging to the reactor may be **triggered** by a present input. Reactions may also be triggered by timers or actions. A reaction may produce outputs, in which case, inputs to which the output is **connected** will become present at the same logical time instant. Outputs, therefore, are **logically**

simultaneous with the inputs that cause them. A reaction may also **schedule** actions which will trigger reactions of the same reactor at a *later* logical time.

In the C target, a timestamp is an unsigned 64-bit integer which, on most platforms, specifies the number of nanoseconds since January 1, 1970. Since a 64-bit number has a limited range, this measure of time instants will overflow in approximately the year 2554. When an LF program starts executing, logical time is (normally) set to the current physical time provided by the operating system. (On some embedded platforms without real-time clocks, it will be set instead to zero.)

At the starting logical time, reactions that specify a **startup** trigger will execute. Also, any reactions that are triggered by a timer with a zero offset will execute. Any outputs produced by these reactions will have the same logical time and will trigger execution of any downstream reactions. Those downstream reactions will be invoked at the same logical time unless some connection to the downstream reactor uses the **after** keyword to specify a time delay. After all reactions at the starting logical time have completed, then time will advance to the logical time of the earliest next event. The earliest next event may be specified by a timer, by an **after** keyword on a connection, or by a logical or physical action. Once logical time advances, any reactions that are triggered by events at that logical will be invoked, as will any reactions triggered by outputs produced by those reactions.

Time in Lingua Franca is actually superdense time, meaning that a logical time may have the same numerical value but also be *strictly later* than another logical time. When an action is scheduled with a delay of zero, it occurs at such a strictly later time, one **microstep** later. See the actions description. The **after** keyword with a time delay of 0 (zero) will also cause the downstream reactions to execute in the next microstep.

At any logical time, if any two reactions belonging to the same reactor are triggered, they will be executed atomically in the order in which they are defined in the reactor. Dependencies across reactors (connections with no **after**) will also result in sequential execution. Specifically, if any reaction of reactor *A* produces an output that triggers a reaction of reactor *B*, then *B*'s reaction will execute only after *A*'s reaction has completed execution. Modulo these two ordering constraints, reactions may execute in parallel on multiple cores or even across networks.

Reactions are given in a **target language**, whereas inputs, outputs, actions, and the dependencies among them are defined in Lingua Franca. LF is, therefore, a kind of **coordination language** rather than a programming language.

Real-Time Systems

Reactions may have delays and deadlines associated with them. This information can be used to perform earliest deadline first (EDF) scheduling. Also, in combination with execution-time analysis of reactions, it should be possible to determine at compile time whether the imposed deadlines can be met, although these analysis tools have not yet been developed. Of particular interest is to

deploy reactors on platforms such as FlexPRET and Patmos, which are designed for predictable timing; execution-time estimates for these architectures will be much tighter than currently possible with ordinary microprocessors.

Schedulability Analysis of LF Programs

- Start with classic schedulability test: critical instant (Liu and Layland)
  - Possible not the worst-case?
  - Question: can not producing an output lead to a timing anomaly?
- Classic schedulability analysis becomes messy when deadline with locks for communication (priority inversion, locking protocols to avoid it, recursive schedulability analysis)
  - We do not use shared data protected by locks +1
- First (pessimistic) approach: all input events and timers fire at the same time, check all execution chains (reactions in actors) need to finish before the actuator deadlines.
- Delays (timer, after, scheduled actions) break the dependency chain
  - Schedulability analysis can be broken up into sub-chains +1
- For now assume no preemptions, this also enables WCET analysis of reactions
- We could also use a big hammer: model the LF program as timed automata and do model checking (e.g., UupAal)

To Do List

Lingua Franca is a work in progress. See our project page for an overview of ongoing and future work.

Tutorial

Lingua Franca (LF) is a polyglot coordination language for concurrent and possibly time-sensitive applications ranging from low-level embedded code to distributed cloud and edge applications. On Oct. 8, 2021, we offered a tutorial on Lingua Franca for the EMSOFT conference, a part of ESWEEK. A video playlist recording is available in six segments, as detailed below.

Useful links:

- This page: [[https://esweek.lf-lang.org/]]
- Complete video playlist
- Part I: Introduction
- Part II: Hello World
- Part III: Target Languages
- Part IV: Basic Concepts
- Part V: Concurrency and Performance
- Part VI: Research Overview
- Slides

Part I: Introduction

This part briefly describes the background of the project and explains how to get started with the software.

Useful links:

- Complete video of part I
- Individual parts of the video:

  | Contents | | ---------- | | Introduction | | Motivation | | Overview of this tutorial | | History of the project | | Participating | | Getting started | | Native releases (Epoch IDE and lfc) | | Virtual Machine with LF pre-installed |

- Virtual machine image: [[https://vm.lf-lang.org/]]: download the `Ubuntu-for-LF.ova` image and import into your favorite virtualization software (e.g., VirtualBox or VMWare Player). Start the VM and run Epoch IDE by clicking on the icon on the left.

- Epoch IDE and lfc command-line compiler: [[https://releases.lf-lang.org/]]. Add `lfc` (and `epoch`) to your `$PATH` environment variable.
- Requirements for each target language: [[https://reqs.lf-lang.org/]] (we use the C target here).

Part II: Hello World

This part introduces the language with a simple example.

Useful links:

- Complete video of part II
- Individual parts of the video: | Contents| |----------| | Open Epoch and create a project| | Hello World| | Adding a timer| | Adding a timeout target property| | Adding state variables | | Creating and connecting multiple reactors | | Parameterized reactors | | LF tour recap | | Diagrams |

Part III: Target Languages

This part focuses on the target languages other than C, namely C++, Python, TypeScript, and Rust.

Useful links:

- Complete video of part III
- Individual parts of the video: | Contents| |----------| | Introduction | | Cpp | | Python | | Python Demo: Piano Synth | | TypeScript | | Rust |

Part IV: Basic Concepts

This part focuses on basic concepts in the language and includes three demos.

Useful links:

- Complete video of part IV
- Individual parts of the video: | Contents| |----------| | Reflex game overview | | Generating the prompts: Basic concepts | | Program control of time: Logical action | | Handling external events: Physical action | | Cycle and causality loop | | Reflex game in Python | | The Rhythm example |

Part V: Concurrency

This part focuses on how the language expresses concurrency, exploits multicore, and supports distributed execution.

Useful links:

- Complete video of part V
- Individual parts of the video: | Contents| |----------| | Introduction | | Banks and Multiports | | Utilizing Multicore | | Tracing | | Performance | | Federated Execution |

Part VI: Research Overview

This part focuses on a few of the research projects that have been stimulated by the Lingua Franca project.

Useful links:

- Complete video of part VI
- Individual parts of the video: | Contents| |----------| | Introduction | | AUTOSAR | | Autoware/Carla | | Bare Iron Platforms | | Modal Models | | Automated Verification | | Secure Federated Execution | | LF Language Server |

Language Specification

A Lingua Franca file, which has a .lf extension, contains the following:

- One **target** specification.
- Zero or more **import** statements.
- One or more **reactor** blocks, which contain **reaction** declarations.

If one of the reactors in the file is designated `main` or `federated`, then the file defines an executable application. Otherwise, it defines one or more library reactors that can be imported into other LF files. For example, an LF file might be structured like this:

```
target C;
main reactor C {
    a = new A();
    b = new B();
    a.y -> b.x;
}
reactor A {
    output y;
    ...
}
reactor B {
    input x;
    ...
}
```

The name of the main reactor (`C` above) is optional. If given, it must match the filename (`C.lf` in the above example).

This example specifies and instantiates two reactors, one of which sends messages to the other. A minimal but complete Lingua Franca file with one reactor is this:

```
target C;
main reactor HelloWorld {
    reaction(startup) {=
        printf("Hello World.\n");
    =}
}
```

See the C target documentation for details about this example.

Target Specification

Every Lingua Franca program begins with a [[target specification]] that specifies the language in which reactions are written. This is also the language of the program(s) generated by the Lingua Franca compiler.

Import Statement

An import statement has the form:

```
import { reactor1, reactor2 as alias2, [...] } from "path";
```

where *path* specifies another Lingua Franca file relative to the location of the current file.

Reactor Block

A **reactor** is a software component that reacts to input events, timer events, and internal events. It has private state variables that are not visible to any other reactor. Its reactions can consist of altering its own state, sending messages to other reactors, or affecting the environment through some kind of actuation or side effect.

The general structure of a reactor block is as follows:

```
reactor name (parameters) {
  state declarations
  method declarations
  input declarations
  output declarations
  timer declarations
  action declarations
  reaction declarations
  contained reactors
   ...
}
```

Parameter, inputs, outputs, timers, actions, and contained reactors all have names, and the names are required to be distinct from one another.

If the **reactor** keyword is preceded by **main**, then this reactor will be instantiated and run by the generated code. If an imported LF file contains a main reactor, that reactor is ignored. Only reactors that not designated `main` are imported. This makes it easy to create a library of reusable reactors that each come with a test case or demonstration in the form of a main reactor.

Parameter Declaration

A reactor class definition can define parameters as follows:

```
reactor ClassName(paramName1:type(expr), paramName2:type(expr)) {
  ...
}
```

Each parameter may have a *type annotation*, written `:type`, and must have a *default value*, written `(expr)`.

The type annotation specifies a type in the target language, which is necessary for some target languages. For instance in C you might write

```
reactor Foo(size: int(100)) {
    ...
}
```

▶ Introduction to basic LF types and expressions... click to expand

Other forms for types and expressions are described in LF types and LF expressions.

How parameters may be used in the body of a reaction depends on the target. For example, in the C target, a `self` struct is provided that contains the parameter values. The following example illustrates this:

```
target C;
reactor Gain(scale:int(2)) {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        SET(y, x->value * self->scale);
    =}
}
```

This reactor, given any input event `x` will produce an output `y` with value equal to the input scaled by the `scale` parameter. The default value of the `scale` parameter is 2, but this can be changed when the `Gain` reactor is instantiated. The `SET()` is the mechanism provided by the C target for

setting the value of outputs. The parameter `scale` and input `x` are just referenced in the C code as shown above.

State Declaration

A state declaration has one of the forms:

> **state** *name*:*type*(*initial_value*);
> **state** *name*(*parameter*);

In the first form, the type annotation is only required in some targets. The initial value may be any expression, including a special initializer forms.

In the second form, the state variable inherits its type from the specified *parameter*, which also provides the initial value for the state variable.

How state variables may be used in the body of a reaction depends on the target. For example, in the C target, a `self` struct is provided that contains the state values. The following example illustrates this:

```
reactor Count {
        output c:int;
        timer t(0, 1 sec);
        state i:int(0);
        reaction(t) -> c {=
                (self->i)++;
                SET(c, self->i);
        =}
}
```

Method Declaration

A method declaration has one of the forms:

> **method** *name*();
> **method** *name*():*type*;
> **method** *name*(*arg1_name*:arg1_type, *arg2_name*:arg2_type, ...); **method** *name*(*arg1_name*:arg1_type, *arg2_name*:arg2_type, ...):*type*;

The first form defines a method with no arguments and no return value. The second form defines a method with the return type *type* but no arguments. The third form defines a method with

arguments given by their name and type, but without a return value. Finally, the fourth form is similar to the third, but adds a return type.

The **method** keywork can optionally be prefixed with the **const** qualifier, which indicates that the method is "read-only". This is relvant for some target languages such as C++.

See the C++ documentation for a usage example.

Input Declaration

An input declaration has the form:

```
input name:type;
```

The `Gain` reactor given above provides an example. The *type* is just like parameter types.

An input may have the modifier **mutable**, as follows:

```
mutable input name:type
```

This is a directive to the code generator indicating that reactions that read this input will also modify the value of the input. Without this modifier, inputs are **immutable**; modifying them is disallowed. The precise mechanism for making use of mutable inputs is target-language specific. See, for example, the C language target.

An input port may have more than one **channel**. See multiports documentation.

Output Declaration

An output declaration has the form:

```
output name:type;
```

The `Gain` reactor given above provides an example. The *type* is just like parameter types.

An output port may have more than one **channel**. See multiports documentation.

Timer Declaration

A timer, like an input and an action, causes reactions to be invoked. Unlike an action, it is triggered automatically by the scheduler. This declaration is used when you want to invoke reactions once at

specific times or periodically. A timer declaration has the form:

> **timer** *name*(*offset*, *period*);

For example,

```
timer foo(10 msec, 100 msec);
```

This specifies a timer named `foo` that will first trigger 10 milliseconds after the start of execution and then repeatedly trigger at intervals of 100 ms. The units are optional, and if they are not included, then the number will be interpreted in a target-dependent way. The units supported are the same as in parameter declarations described above.

The times specified are logical times. Specifically, if two timers have the same *offset* and *period*, then they are logically simultaneous. No observer will be able to see that one timer has triggered and the other has not. Even though these are logical times, the runtime system will make an effort to align those times to physical times. Such alignment can never be perfect, and its accuracy will depend on the execution platform.

Both arguments are optional, with both having default value zero. An *offset* of zero or greater specifies the minimum time delay between the time at the start of execution and when the action is triggered. The *period* is zero or greater, where a value of zero specifies that the reactions should be triggered exactly once, whereas a value greater than zero specifies that they should be triggered repeatedly with the period given.

To cause a reaction to be invoked at the start of execution, a special **startup** trigger is provided:

```
reactor Foo {
    reaction(startup) {=
        ... perform initialization ...
    =}
}
```

The **startup** trigger is equivalent to a timer with no *offset* or *period*.

Action Declaration

An **action**, like an input, can cause reactions to be invoked. Whereas inputs are provided by other reactors, actions are scheduled by this reactor itself, either in response to some observed external event or as a delayed response to some input event. The action can be scheduled by a reactor by invoking a **schedule** function in a reaction or in an asynchronous callback function.

An action declaration is either physical or logical:

```
physical action name(min_delay, min_spacing, policy):type;
logical action name(min_delay, min_spacing, policy):type;
```

The *min_delay*, *min_spacing*, and *policy* are all optional. If only one argument is given in parentheses, then it is interpreted as an *min_delay*, if two are given, then they are interpreted as *min_delay* and *min_spacing*, etc. The *min_delay* and *min_spacing* have to be a time value. The *policy* argument is a string that can be one of the following: `'defer'` (default), `'drop'`, or `'replace'`.

An action will trigger at a logical time that depends on the arguments given to the schedule function, the *min_delay*, *min_spacing*, and *policy* arguments above, and whether the action is physical or logical.

If the **logical** keyword is given, then the tag assigned to the event resulting from a call to **schedule** function is computed as follows. First, let *t* be the *current logical time*. For a logical action, the `schedule` function must be invoked from within a reaction (synchronously), so *t* is just the logical time of that reaction.

The (preliminary) tag of the action is then just *t* plus *min_delay* plus the *offset* argument to **schedule** function.

If the **physical** keyword is given, then the physical clock on the local platform is used as the timestamp assigned to the action. Moreover, for a physical action, unlike a logical action, the `schedule` function can be invoked from outside of any reaction (asynchronously), e.g. from an interrupt service routine or callback function.

If a *min_spacing* has been declared, then a minimum distance between the tags of two subsequently scheduled events on the same action is enforced. If the preliminary tag is closer to the tag of the previously scheduled event (if there is one), then *policy* determines how the given constraints is enforced.

- `'drop'`: the new event is dropped and `schedule` returns without having modified the event queue.
- `'replace'`: the payload of the new event is assigned to the preceding event if it is still pending in the event queue; no new event is added to the event queue in this case. If the preceding event has already been pulled from the event queue, the default `'defer'` policy is applied.
- `'defer'`: the event is added to the event queue with a tag that is equal to earliest time that satisfies the minimal spacing requirement. Assuming the tag of the preceding event is *t_prev*, then the tag of the new event simply becomes *t_prev* + *min_spacing*.

Note that while the `'defer'` policy is conservative in the sense that it does not discard events, it could potentially cause an unbounded growth of the event queue.

In all cases, the logical time of a new event will always be strictly greater than the logical time at which it is scheduled by at least one microstep (see the Time section).

The default *min_delay* is zero. The default *min_spacing* is undefined (meaning that no minimum spacing constraint is enforced). If a `min_spacing` is defined, it has to be strictly greater than zero, and greater than or equal to the time precision of the target (for the C target, it is one nanosecond).

The *min_delay* parameter in the **action** declaration is static (set at compile time), while the *offset* parameter given to the schedule function may be dynamically set at runtime. Hence, for static analysis and scheduling, the **action**'s' *min_delay* parameter can be assumed to be a *minimum delay* for analysis purposes.

Discussion

Logical actions are used to schedule events at a future logical time relative to the current logical time. Physical time is ignored. They must be scheduled within reactions, and the timestamp of the scheduled event will be relative to the current logical time of the reaction that schedules them. It is an error to schedule a logical action asynchronously, outside of the context of a reaction. Asynchronous actions are required to be **physical**.

Physical actions are typically used to assign timestamps to externally triggered events, such as the arrival of a network message or the acquisition of sensor data, where the time at which these external events occurs is of interest. There are (at least) three interesting use cases:

1. An asynchronous event, such as a callback function or interrupt service routine (ISR), is invoked at a physical time *t* and schedules an action with timestamp $T = t$. To get this behavior, just set the physical action to have *min_delay* = 0 and call the schedule function with *offset* = 0. The *min_spacing* can be useful here to prevent these external events from overwhelming the software system.
2. A periodic task that is occasionally modified by a sporadic sensor. In this case, you can set *min_delay* = *period* and call schedule with *offset* = 0. The resulting timestamp of the sporadic sensor event will always align with the periodic events. This is similar to periodic polling, but without the overhead of polling the sensor when nothing interesting is happening.
3. You can impose a minimum physical time delay between an event's occurrence, such as a push of a button, and system response by adjusting the *offset*.

Actions With Values

If an action is declared with a *type*, then it can carry a **value**, a data value passed to the **schedule** function. This value will be available to any reaction that is triggered by the action. The specific mechanism, however, is target-language dependent. See the C target for an example.

Reaction Declaration

A reaction is defined within a reactor using the following syntax:

```
reaction(triggers) uses -> effects {=
  ... target language code ...
=}
```

The *uses* and *effects* fields are optional. A simple example appears in the "hello world" example given above:

```
reaction(t) {=
    printf("Hello World.\n");
=}
```

In this example, `t` is a **trigger** (a timer named `t`). When that timer fires, the reaction will be invoked. Triggers can be timers, inputs, outputs of contained reactors, or actions. A comma-separated list of triggers can be given, in which case any of the specified triggers can trigger the reaction. If, at any logical time instant, more than one of the triggers fires, the reaction will nevertheless be invoked only once.

The *uses* field specifies inputs that the reaction observes but that do not trigger the reaction. This field can also be a comma-separated list of inputs. Since the input does not trigger the reaction, the body of the reaction will normally need to test for presence of the input before using it. How to do this is target specific. See how this is done in the C target.

The *effects* field, occurring after the right arrow, declares which outputs and actions the target code *may* produce or schedule. The *effects* field may also specify inputs of contained reactors, provided that those inputs do not have any other sources of data. These declarations make it *possible* for the reaction to send outputs or enable future actions, but they do not *require* that the reaction code do that.

Target Code

The body of the reaction is code in the target language surrounded by `{=` and `=}`. This code is not parsed by the Lingua Franca compiler. It is used verbatim in the program that is generated.

The target provides language-dependent mechanisms for referring to inputs, outputs, and actions in the target code. These mechanisms can be different in each target language, but all target languages provide the same basic set of mechanisms. These mechanisms include:

- Obtaining the current logical time (logical time does not advance during the execution of a reaction, so the execution of a reaction is logically instantaneous).
- Determining whether inputs are present at the current logical time and reading their value if they are. If a reaction is triggered by exactly one input, then that input will always be present. But if there are multiple triggers, or if the input is specified in the *uses* field, then the input may not be present when the reaction is invoked.
- Setting output values. Reactions in a reactor may set an output value more than once at any instant of logical time, but only the last of the values set will be sent on the output port.
- Scheduling future actions.

In the C target, for example, the following reactor will add two inputs if they are present at the time of a reaction:

```
reactor Add {
    input in1:int;
    input in2:int;
    output out:int;
    reaction(in1, in2) -> out {=
        int result = 0;
        if (in1->is_present) result += in1->value;
        if (in2->is_present) result += in2->value;
        SET(out, result);
    =}
}
```

See the C target for an example of how these things are specified in C.

NOTE: if a reaction fails to test for the presence of an input and reads its value anyway, then the result it will get is undefined and may be target dependent. In the C target, as of this writing, the value read will be the most recently seen input value, or, if no input event has occurred at an earlier logical time, then zero or NULL, depending on the datatype of the input. In the TS target, the value will be **undefined**, a legitimate value in TypeScript.

Scheduling Future Reactions

Each target language provides some mechanism for scheduling future reactions. Typically, this takes the form of a `schedule` function that takes as an argument an action, a time interval, and (perhaps optionally), a payload. For example, in the C target, in the following program, each reaction to the timer `t` schedules another reaction to occur 100 msec later:

```
target C;
main reactor Schedule {
    timer t(0, 1 sec);
    logical action a;
    reaction(t) -> a {=
        schedule(a, MSEC(100));
    =}
    reaction(a) {=
        printf("Nanoseconds since start: %lld.\n", get_elapsed_logical_time
    =}
}
```

When executed, this will produce the following output:

```
Start execution at time Sun Aug 11 04:11:57 2019
plus 919310000 nanoseconds.
Nanoseconds since start: 100000000.
Nanoseconds since start: 1100000000.
Nanoseconds since start: 2100000000.
...
```

This action has no datatype and carries no value, but, as explained below, an action can carry a value.

Asynchronous Callbacks

In targets that support multitasking, the `schedule` function, which schedules future reactions, may be safely invoked on a **physical action** in code that is not part of a reaction. For example, in the multithreaded version of the C target, `schedule` may be invoked in an interrupt service routine. The reaction(s) that are scheduled are guaranteed to occur at a time that is strictly larger than the current logical time of any reactions that are being interrupted.

Superdense Time

Lingua Franca uses a concept known as **superdense time**, where two time values that appear to be the same are not logically simultaneous. At every logical time value, for example midnight on January 1, 1970, there exist a logical sequence of **microsteps** that are not simultaneous. The Microsteps example illustrates this:

```
target C;
reactor Destination {
    input x:int;
    input y:int;
    reaction(x, y) {=
        printf("Time since start: %lld.\n", get_elapsed_logical_time());
        if (x->is_present) {
            printf("  x is present.\n");
        }
        if (y->is_present) {
            printf("  y is present.\n");
        }
    =}
}
main reactor Microsteps {
    timer start;
    logical action repeat;
    d = new Destination();
    reaction(start) -> d.x, repeat {=
        SET(d.x, 1);
        schedule(repeat, 0);
    =}
    reaction(repeat) -> d.y {=
        SET(d.y, 1);
    =}
}
```

The `Destination` reactor has two inputs, `x` and `y`, and it simply reports at each logical time where either is present what is the logical time and which is present. The `Microsteps` reactor initializes things with a reaction to the one-time timer event `start` by sending data to the `x` input of `Destination`. It then schedules a `repeat` action.

Note that time delay in the call to `schedule` is zero. However, any reaction scheduled by `schedule` is required to occur **strictly later** than current logical time. In Lingua Franca, this is handled by scheduling the `repeat` reaction to occur one **microstep** later. The output printed, therefore, will look like this:

```
Time since start: 0.
  x is present.
Time since start: 0.
  y is present.
```

Note that the numerical time reported by `get_elapsed_logical_time()` has not advanced in the second reaction, but the fact that `x` is not present in the second reaction proves that the first reaction and the second are not logically simultaneous. The second occurs one microstep later.

Note that it is possible to write code that will prevent logical time from advancing except by microsteps. For example, we could replace the reaction to `repeat` in `Main` with this one:

```
    reaction(repeat) -> d.y, repeat {=
        SET(d.y, 1);
        schedule(repeat, 0);
    =}
```

This would create what is known as a **stuttering Zeno** condition, where logical time cannot advance. The output will be an unbounded sequence like this:

```
Time since start: 0.
  x is present.
Time since start: 0.
  y is present.
Time since start: 0.
  y is present.
Time since start: 0.
  y is present.
...
```

Startup and Shutdown Reactions

Two special triggers are supported, **startup** and **shutdown**. A reaction that specifies the **startup** trigger will be invoked at the start of execution of the model. The following two syntaxes have exactly the same effect:

```
    reaction(startup) {= ... =}
```

and

```
    timer t;
    reaction(t) {= ... =}
```

In other words, **startup** is a timer that triggers once at the first logical time of execution. As with any other reaction, the reaction can also be triggered by inputs and can produce outputs or schedule actions.

The **shutdown** trigger is slightly different. A shutdown reaction is specified as follows:

```
   reaction(shutdown) {= ... =}
```

This reaction will be invoked when the program terminates normally (there are no more events, some reaction has called a `request_stop()` utility provided in the target language, or the execution was specified to last a finite logical time). The reaction will be invoked at a logical time one microstep *later* than the last logical time of the execution. In other words, the presence of this reaction means that the program will execute one extra logical time cycle beyond what it would have otherwise, and that logical time is one microstep later than what would have otherwise been the last logical time.

If the reaction produces outputs, then downstream reactors will also be invoked at that later logical time. If the reaction schedules future reactions, those will be ignored. After the completion of this final logical time cycle, one microstep later than the normal termination, the program will exit.

Contained Reactors

Reactors can contain instances of other reactors defined in the same file or in an imported file. Assuming the above Count reactor is stored in a file Count.lf, then CountTest is an example that imports and instantiates it to test the reactor:

```
target C;
import Count.lf;
reactor Test {
    input c:int;
    state i:int(0);
    reaction(c) {=
        printf("Received %d.\n", c->value);
        (self->i)++;
        if (c->value != self->i) {
            printf("ERROR: Expected %d but got %d\n.", self->i, c->value);
            exit(1);
        }
    =}
    reaction(shutdown) {=
        if (self->i != 4) {
            printf("ERROR: Test should have reacted 4 times, but reacted %d
            exit(2);
        }
    =}
}

main reactor CountTest {
    count = new Count();
    test = new Test();
    count.out -> test.c;
}
```

An instance is created with the syntax:

```
instance_name = new class_name(parameters);
```

A bank with several instances can be created in one such statement, as explained in the banks of reactors documentation.

The *parameters* argument has the form:

```
parameter1_name = parameter1_value, parameter2_name = parameter2_value, ...
```

Connections between ports are specified with the syntax:

```
output_port -> input_port
```

where the ports are either *instance_name.port_name* or just *port_name*, where the latter form denotes a port belonging to the reactor that contains the instances.

Physical Connections

A subtle and rarely used variant is a **physical connection**, denoted `~>`. In such a connection, the logical time at the recipient is derived from the local physical clock rather than being equal to the logical time at the sender. The physical time will always exceed the logical time of the sender, so this type of connection incurs a nondeterministic positive logical time delay. Physical connections are useful sometimes in [[Distributed-Execution]] in situations where the nondeterministic logical delay is tolerable. Such connections are more efficient because timestamps need not be transmitted and messages do not need to flow through through a centralized coordinator (if a centralized coordinator is being used).

Connections with Delays

Connections may include a **logical delay** using the **after** keyword, as follows:

```
output_port -> input_port after 10 msec
```

This means that the logical time of the message delivered to the *input_port* will be 10 milliseconds larger than the logical time of the reaction that wrote to *output_port*. If the time value is greater than zero, then the event will appear at microstep 0. If it is equal to zero, then it will appear at the current microstep plus one.

When there are multiports or banks of reactors, several channels can be connected with a single connection statement. See Multiports and Banks of Reactors.

The following example defines a reactor that adds a counting sequence to its input. It uses the above Count and Add reactors (see Hierarchy2):

```
import Count.lf;
import Add.lf;
reactor AddCount {
    input in:int;
    output out:int;
    count = new Count();
    add = new Add();
    in -> add.in1;
    count.out -> add.in2;
    add.out -> out;
}
```

A reactor that contains other reactors may, within a reaction, send data to the contained reactor. The following example illustrates this (see SendingInside):

```
target C;
reactor Printer {
        input x:int;
        reaction(x) {=
                printf("Inside reactor received: %d\n", x->value);
        =}
}
main reactor SendingInside {
        p = new Printer();
        reaction(startup) -> p.x {=
                SET(p.x, 1);
        =}
}
```

Running this will print:

```
Inside reactor received: 1
```

Deadlines

Lingua Franca includes a notion of a **deadline**, which is a relation between logical time and physical time. Specifically, a program may specify that the invocation of a reaction must occur within some physical-time interval of the logical timestamp of the message. If a reaction is invoked at logical time 12 noon, for example, and the reaction has a deadline of one hour, then the reaction is required to be invoked before the physical-time clock of the execution platform reaches 1 PM. If the deadline is violated, then the specified deadline handler is invoked instead of the reaction. For example (see Deadline):

```
reactor Deadline() {
    input x:int;
    output d:int; // Produced if the deadline is violated.
    reaction(x) -> d {=
        printf("Normal reaction.\n");
    =} deadline(10 msec) {=
        printf("Deadline violation detected.\n");
        SET(d, x->value);
    =}
```

This reactor specifies a deadline of 10 milliseconds (this can be a parameter of the reactor). If the reaction to `x` is triggered later in physical time than 10 msec past the timestamp of `x`, then the second body of code is executed instead of the first. That second body of code has access to anything the first body of code has access to, including the input `x` and the output `d`. The output can be used to notify the rest of the system that a deadline violation occurred.

The amount of the deadline, of course, can be given by a parameter.

A sometimes useful pattern is when a container reactor reacts to deadline violations in a contained reactor. The DeadlineHandledAbove example illustrates this:

```
target C;
reactor Deadline() {
    input x:int;
    output deadline_violation:bool;
    reaction(x) -> deadline_violation {=
        ... normal code to execute ...
    =} deadline(100 msec) {=
        printf("Deadline violation detected.\n");
        SET(deadline_violation, true);
    =}
}
main reactor DeadlineHandledAbove {
    d = new Deadline();
    ...
    reaction(d.deadline_violation) {=
        ... handle the deadline violation ...
    =}
}
```

Comments

Lingua Franca files can have C/C++/Java-style comments and/or Python-style comments. All of the following are valid comments:

```
// Single-line C-style comment.
/*
   Multi-line C-style comment.
 */
# Single-line Python-style comment.
'''
   Multi-line Python-style comment.
'''
```

Appendix: LF types

Type annotations may be written in many places in LF, including parameter declarations, state variable declarations, input and output declarations. In some targets, they are required, because the target language requires them too.

Assigning meaning to type annotations is entirely offloaded to the target compiler, as LF does not feature a type system (yet?). However, LF's syntax for types supports a few idioms that have target-specific meaning. Types may have the following forms:

- the **time** type is reserved by LF, its values represent time durations. The **time** type accepts *time expressions* for values, eg `100 msec`, or `0` (see Basic expressions for a reference).
- identifiers are valid types (eg `int`, `size_t`), and may be followed by type arguments (eg `vector<int>`).
- the syntactic forms `type[]` and `type[integer]` correspond to target-specific array types. The second form is available only in languages which support fixed-size array types (eg in C++, `std::array<5>`).
- the syntactic form `{= some type =}` allows writing an arbitrary type as target code. This is useful in target languages which have complex type grammar (eg in TypeScript, `{= int | null =}`).

Also note that to use strings conveniently in the C target, the "type" `string` is an alias for `{=char*=}`.

(Types ending with a `*` are treated specially by the C target. See Sending and Receiving Arrays and Structs in the C target documentation.)

Appendix: LF expressions

A subset of LF syntax is used to write *expressions*, which represent target language values. Expressions are used in state variable initializers, default values for parameters, and parameter assignments.

Expressions in LF support only simple forms, that are intended to be common across languages. Their precise meaning (eg the target language types they are compatible with) is target-specific and not specified here.

Basic expressions

The most basic expression forms, which are supported by all target languages, are the following:

- Literals:
  - Numeric literals, eg `1`, `-120`, `1.5`. Note that the sign, if any, is part of the literal and must not be separated by whitespace.
  - String literals, eg `"abcd"`. String literals always use double-quotes, even in languages which support other forms (like Python).
  - Character literals. eg `'a'`. Single-quoted literals must be exactly one character long --even in Python.
  - Boolean literals: `true`, `false`, `True`, `False`. The latter two are there for Python.
- Parameter references, which are simple identifiers (eg `foo`). Any identifier in expression position must refer to a parameter of the enclosing reactor.
- Time values, eg `1 msec` or `10 seconds`. The syntax of time values is `integer time_unit`, where `time_unit` is one of the following
  - **nsec**: nanoseconds
  - **usec**: microseconds
  - **msec**: milliseconds
  - **sec** or **second**: seconds
  - **minute**: 60 seconds
  - **hour**: 60 minutes
  - **day**: 24 hours
  - **week**: 7 days

Each of these units also support a pluralized version (eg `nsecs`, `minutes`, `days`), which means the same thing.

The time value `0` may have no unit. Except in this specific case, the unit is always required.

Time values are compatible with the `time` type.

- Escaped target-language expression, eg `{= foo() =}`. This syntax is used to write any expression which does not fall into one of the other forms described here. The contents are not parsed and are used verbatim in the generated file.

  The variables in scope are target-specific.

Complex expressions

Some targets may make use of a few other syntactic forms for expressions. These syntactic forms may be acribed a different meaning by different targets, to keep the source language close in meaning to the target language.

We describe here these syntactic forms and what meaning they have in each target.

- Bracket-list syntax, eg `[1, 2, 3]`. This syntax is used to create a list in Python. It is not supported by any other target at the moment.

  ```
  state x([1,2,3])
  ```

Initializer pseudo-expressions

Some "expression" forms are only acceptable as the initializer of a state variable or parameter, but not in other places (like inside a list expression). These are

- Tuple syntax, eg `(1, 2, 3)`. This syntax is used:

  - in the Python target, to create a tuple value. Tuples are different from lists in that they are immutable.
  - in C++, to pass arguments to a constructor:

    ```
    state x: int[](1,2);
    ```

    In that example, the initializer expression is translated to `new std::vector(1,2)`. See also C++ target documentation.
  - in C and all other targets, to create a target-specific array value. In the Python target, this is accomplished by the bracket-list syntax `[1,2,3]` instead. Note that to create a zero- or one-

element array, fat braces are usually required. For instance in C:

```
state x: int[](1,2,3); // creates an int array, basically `int x[] = {1
state x: int[](1);     // `int x[] = 1;` - type error!
state x: int[]({= {1} =})  // one element array: `int x[] = {1};`
```

- Brace-list syntax, eg `{1, 2, 3}`. This syntax is at the moment only supported by the C++ target. It's used to initialize a vector with the initializer list syntax instead of a constructor call.

Multiports and Banks

This page describes Lingua Franca language constructs support more scalable programs by providing a compact syntax for ports that can send or receive over multiple channels (called **multiports**) and multiple instances of a reactor class (called a **bank of reactors**). The examples given below include syntax of the C target for accessing the ports. Other targets use different syntax with target code, within the delimiters `{= ... =}`, but use the same syntax outside those delimiters.

Multiports

To declare an input or output port to be a **multiport**, use the following syntax:

```
input[width] name:type; output[width] name:type;
```

where *width* is a positive integer. This can be given either as an integer literal or a parameter name. For targets that allow dynamic parametrization at runtime (like the C++ target), width can also be given by target code enclosed in `{=...=}`.

For example, (see MultiportToMultiport):

```
target C;
reactor Source {
    output[4] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
reactor Destination {
    input[4] in:int;
    reaction(in) {=
        int sum = 0;
        for (int i = 0; i < in_width; i++) {
            if (in[i]->is_present) sum += in[i]->value;
        }
        printf("Sum of received: %d.\n", sum);
    =}
}
main reactor MultiportToMultiport {
    a = new Source();
    b = new Destination();
    a.out -> b.in;
}
```

The `Source` reactor has a four-way multiport output and the `Destination` reactor has a four-way multiport input. These channels are connected all at once on one line, the second line from the last. Running this program produces:

```
Sum of received: 6.
```

NOTE: In `Destination`, the reaction is triggered by `in`, not by some individual channel of the multiport input. Hence, it is important when using multiport inputs to test for presence of the input on each channel, as done above with the syntax `if (in[i]->is_present) ...`. An event on any one of the channels is sufficient to trigger the reaction.

Sending and Receiving Via a Multiport

The source reactor specifies `out` as an effect of its reaction using the syntax `-> out`. This brings into scope of the reaction body a way to access the width of the port and a way to write to each channel of the port. It is also possible to test whether a previous reaction has set an output value and to read what that value is. The exact syntax for this depends on the target language. In the C

target, the width is accessed with the variable `out_width`, and `out[i]` references the output channel to write to using the `SET` macro, as shown above. In addition, `out[i]->is_present` and `out[i]->value` are defined. For example, if we modify the above reaction as follows:

```
reactor Source {
    output[4] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            printf("Before SET, out[%d]->is_present has value %d\n", i, out
            SET(out[i], i);
            printf("AFTER set, out[%d]->is_present has value %d\n", i, out[
            printf("AFTER set, out[%d]->value has value %d\n", i, out[i]->v
        }
    =}
}
```

then we get the output:

```
Before SET, out[0]->is_present has value 0
AFTER set, out[0]->is_present has value 1
AFTER set, out[0]->value has value 0
Before SET, out[1]->is_present has value 0
AFTER set, out[1]->is_present has value 1
AFTER set, out[1]->value has value 1
Before SET, out[2]->is_present has value 0
AFTER set, out[2]->is_present has value 1
AFTER set, out[2]->value has value 2
Before SET, out[3]->is_present has value 0
AFTER set, out[3]->is_present has value 1
AFTER set, out[3]->value has value 3
Sum of received: 6.
```

If you access `out[i]->value` before any value has been set, the result is undefined.

In the Python target, multiports can be iterated on in a for loop (e.g., `for p in out`) or enumerated (e.g., `for i, p in enumerate(out)`) and the length of the multiport can be obtained by using the `len()` (e.g., `len(out)`) expression.

Parameterized Widths

The width of a port may be given by a parameter. For example, the above `Source` reactor can be rewritten

```
reactor Source(width:int(4)) {
    output[width] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
```

In some targets such as the C++ target, parameters to the main reactor can be overwritten at the command line interface, allowing for dynamically scalable applications.

Connecting Reactors with Different Widths

Assume that the `Source` and `Destination` reactors above both use a parameter `width` to specify the width of their ports. Then the following connection is valid (see MultiportToMultiport2)

```
main reactor MultiportToMultiport2 {
    a1 = new Source(width = 3);
    a2 = new Source(width = 2);
    b = new Destination(width = 5);
    a1.out, a2.out -> b.in;
}
```

The first three ports of `b` will received input from `a1`, and the last two ports will receive input from `a2`. Parallel composition can appear on either side of a connection. For example:

```
    a1.out, a2.out -> b1.out, b2.out, b3.out;
```

If the total width on the left does not match the total width on the right, then a warning is issued. If the left side is wider than the right, then output data will be discarded. If the right side is wider than the left, then inputs channels will be absent.

Any given port can appear only once on the right side of the `->` connection operator, so all connections to a multiport destination must be made in one single connection statement.

Banks of Reactors

Using a similar notation, it is possible to create a bank of reactors. For example, we can create a bank of four instances of `Source` and four instances of `Destination` and connect them as follows (see MultiportToBankMultiport):

```
main reactor BankToBankMultiport {
    a = new[4] Source();
    b = new[4] Destination();
    a.out -> b.in;
}
```

If the `Source` and `Destination` reactors have multiport inputs and outputs, as in the examples above, then a warning will be issued if the total width on the left does not match the total width on the right. For example, the following is balanced:

```
main reactor BankToBankMultiport {
    a = new[3] Source(width = 4);
    b = new[4] Destination(width = 3);
    a.out -> b.in;
}
```

There will be three instances of `Source`, each with an output of width four, and four instances of `Destination`, each with an input of width 3, for a total of 12 connections.

To distinguish the instances in a bank of reactors, the reactor can define a parameter called **bank_index** with any type that can be assigned a non-negative integer value (in C, for example, `int`, `size_t`, or `uint32_t` will all work). If such a parameter is defined for the reactor, then when the reactor is instanced in a bank, each instance will be assigned a number between 0 and *n*-1, where *n* is the number of reactor instances in the bank. For example, the following source reactor increments the output it produces by the value of `bank_index` on each reaction to the timer (see BankToBank):

```
reactor Source(
    bank_index:int(0)
) {
    timer t(0, 200 msec);
    output out:int;
    state s:int(0);
    reaction(t) -> out {=
        SET(out, self->s);
        self->s += self->bank_index;
    =}
}
```

The width of a bank may also be given by a parameter, as in

```
main reactor BankToBankMultiport(
    source_bank_width:int(3),
    destination_bank_width:int(4)
) {
    a = new[source_bank_width] Source(width = 4);
    b = new[destination_bank_width] Destination(width = 3);
    a.out -> b.in;
}
```

Contained Banks

Banks of reactors can be nested. For example, note the following program:

```
target C;
reactor Child (
    bank_index:int(0)
) {
    reaction(startup) {=
        info_print("My bank index is %d.", self->bank_index);
        =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child();
}
main reactor {
    p = new[2] Parent();
}
```

In this program, the `Parent` reactor contains a bank of `Child` reactor instances with a width of 2. In the main reactor, a bank of `Parent` reactors is instantiated with a width of 2, therefore, creating 4 `Child` instances in the program in total. The output of this program will be:

```
My bank index is 0.
My bank index is 1.
My bank index is 0.
My bank index is 1.
```

Moreover, the bank index of a container (parent) reactor can be passed down to contained (child) reactors. For example, note the following program:

```
target C;
reactor Child (
    bank_index:int(0),
    parent_bank_index:int(0)
) {
    reaction(startup) {=
        info_print(
            "My parent's bank index is %d.",
            self->parent_bank_index
            );
        =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
}
main reactor {
    p = new[2] Parent();
}
```

In this example, the bank index of the `Parent` reactor is passed to the `parent_bank_index` parameter of the `Child` reactor instances. The output from this program will be:

```
My parent's bank index is 0.
My parent's bank index is 1.
My parent's bank index is 1.
My parent's bank index is 0.
```

Finally, members of contained banks of reactors can be individually addressed in the body of reactions of the parent reactor if their input/output port appears in the reaction signature. For example, note the following program:

```
target C;
reactor Child (
    bank_index:int(0),
    parent_bank_index:int(0)
) {
    output out:int;
    reaction(startup) -> out {=
        SET(out, self->parent_bank_index + self->bank_index);
        =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
    reaction(c.out) {=
        for (int i=0; i < c_width; i++) {
            info_print("Received %d from child %d.", c[i].out->value, i);
        }
    =}
}
main reactor {
    p = new[2] Parent();
}
```

Note the usage of `c_width`, which holds the width of the `c` bank of reactors. Note that this syntax is target-specific. For example, in the Python target, `len(c)` can be used to get the width of the bank, and `for p in c` or `for (i, p) in enumerate(c)` can be used to iterate over the banks.

Combining Banks and Multiports

Banks of reactors may be combined with multiports (see MultiportToBank):

```
reactor Source {
    output[3] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
reactor Destination(
    bank_index:int(0)
) {
    input in:int;
    reaction(in) {=
        printf("Destination %d received %d.\n", self->bank_index, in->value
    =}
}

main reactor MultiportToBank {
    a = new Source();
    b = new[3] Destination();
    a.out -> b.in;
}
```

The three outputs from the `Source` instance `a` will be sent, respectively, to each of three instances of `Destination`, `b[0]`, `b[1]`, and `b[2]`.

The reactors in a bank may themselves have multiports. In all cases, the number of ports on the left of a connection must match the number on the right, unless the ones on the left are iterated, as explained next.

Broadcast Connections

Occasionally, you will want to have fewer ports on the left of a connection and have their outputs used repeatedly to broadcast to the ports on the right. In the ThreadedThreaded example, the outputs from an ordinary port are broadcast to the inputs of all instances of a bank of reactors:

```
reactor Source {
        output out:int;
        reaction(startup) -> out {=
                SET(out, 42);
        =}
}
reactor Destination {
        input in:int;
        reaction(in) {=
                ...
        =}
}
main reactor ThreadedThreaded(width:int(4)) {
        a = new Source();
        d = new[width] Destination();
        (a.out)+ -> d.in;
}
```

The syntax `(a.out)+` means "repeat the output port `a.out` one or more times as needed to supply all the input ports of `d.in`." The content inside the parentheses can be a comma-separated list of ports, the ports inside can be ordinary ports or multiports, and the reactors inside can be ordinary reactors or banks of reactors. In all cases, the number of ports inside the parentheses on the left must divide the number of ports on the right.

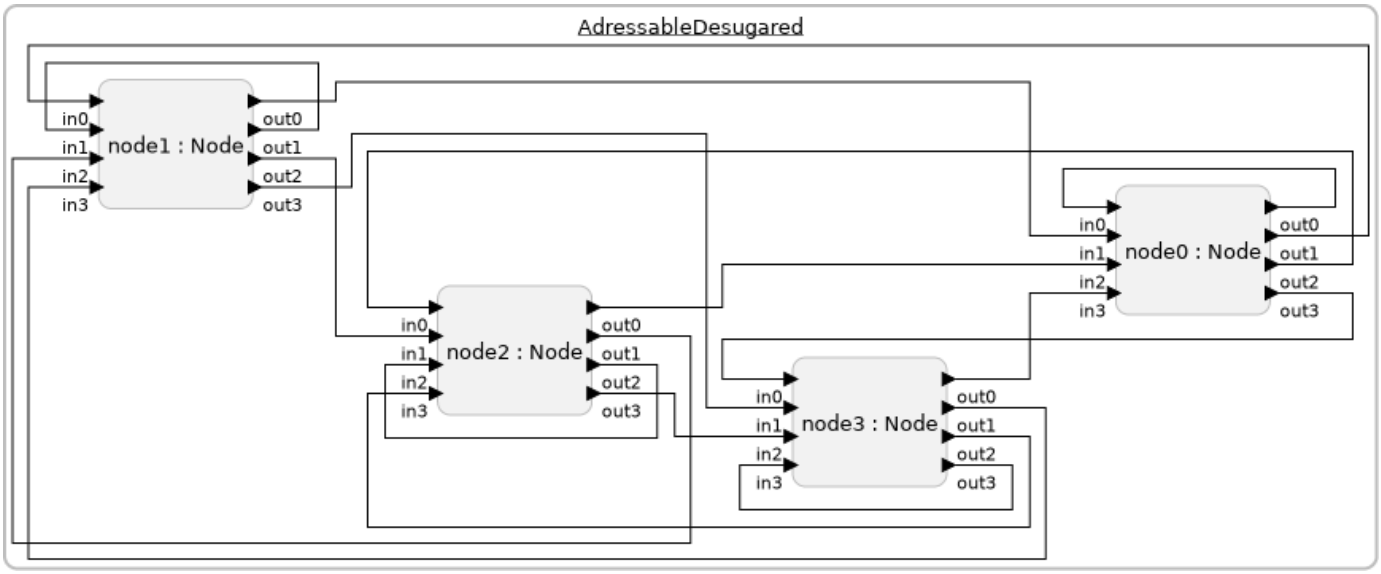
Interleaved Connections

Sometimes, we don't want to broadcast messages to all reactors, but need more fine-grained control as to which reactor within a bank receives a message. If we have separate source and destination reactors, this can be done by combining multiports and banks as was shown in Combining Banks and Multiports. Setting a value on the index N of the output multiport, will result in a message to the Nth reactor instance within the destianation bank. However, this pattern gets slightly more complicated, if we want to exchange addressable messages between instances of the same bank. This pattern is shown in the FullyConnected_01_Addressable example, which is simplified below:

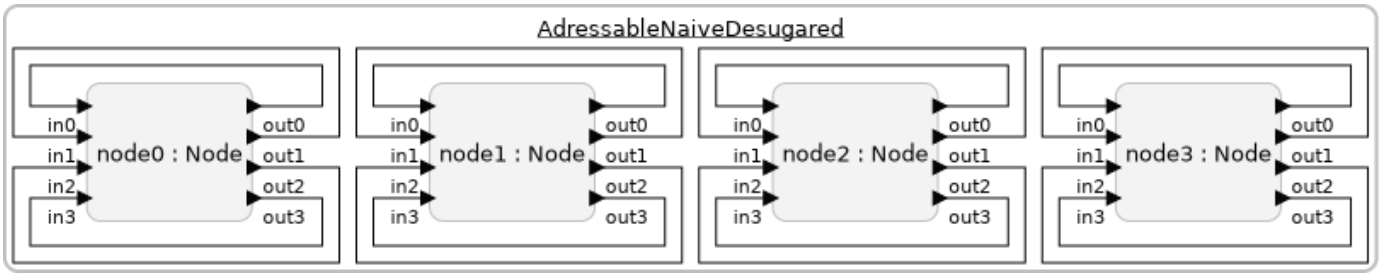
```
reactor Node(
    num_nodes: size_t(4),
    bank_index: int(0)
) {
    input[num_nodes] in: int;
    output[num_nodes] out: int;

    reaction (startup) -> out {=
        SET(out[1], 42);
    =}

    reaction (in) {=
        ...
    =}
}

main reactor(num_nodes: size_t(4)) {
    nodes = new[num_nodes] Node(num_nodes=num_nodes);
    nodes.out -> interleaved(nodes.in);
}
```

In the above program, four instance of `Node` are created, and, at startup, each instance sends 42 to its second (index 1) output channel. The result is that the second bank member (`bank_index` 1) will receive the number 42 on each input channel of its multiport input. The 0-th channel will receive from `bank_index` 0, the 1-th channel from `bank_index` 1, etc. In effect, the choice of output channel specifies the destination reactor in the bank, and the input channel specifies the source reactor.

This style of connection is accomplished using the new keyword **interleaved** in the connection. Normally, a port reference such as `nodes.out` where `nodes` is a bank and `out` is a multiport, would list all the individual ports by first iterating over the banks and then the ports. If we consider the tuple (b,p) to denote the index b within the bank and the index p within the multiport, then the following list is created: (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3). However, if we use `interleaved(nodes.out)` instead, the connection logic will iterate over the ports first and then the banks, creating the following list: (0,0), (1,0), (2,0), (3,0), (0,1), (1,1), (2,1), (3,1), (0,2), (1,2), (2,2), (3,2), (0,3), (1,3), (2,3), (3,3). By combining a normal port reference with a interleaved reference, we can construct a fully connected network. The figure below visualizes this pattern and shows a desugared version constructed without banks or multiports:



If we would use a normal connection instead `nodes.out -> nodes.in;`, then the following pattern would be created:



Effectively, this connects each reactor instance to itself, which isn't very useful.

Downloading and Building

Using Lingua Franca

To get started with Lingua Franca immediately, download `Epoch` (our IDE) and/or `lfc` (our command-line compiler) from one of the following releases:

- Nightly Build
- Version 0.1.0-alpha

IMPORTANT NOTE: MacOS will report that `lflang.app` is broken because it was not signed. To execute it, please run

```
xattr -cr epoch.app
```

first on the command line. Eventually, we will provide a signed download.

If you plan to just use the command-line compiler, you may want a language plugin for Vim and Neovim. See the installation instructions.

Working from the git Repository

If you plan to contribute to Lingua Franca, or if you want to keep up to date as the project evolves, you will need to work from the git repository on GitHub. There are several ways to do this:

1. **Recommended**: Oomph setup for Eclipse: Follow the [[Developer Eclipse setup with Oomph]] instructions.
2. You can [[Clone the Repository]] and build manually using gradle or maven.

- Gradle: `./gradlew assemble` (the `build` also performs tests, which takes a long time)
- Maven: `mvn compile` (you need to install Maven first)

3. Some of code generator components are written in Kotlin, which is not supported by Eclipse. If you want a Kotlin-friendly developer environment using IntelliJ, you can follow the [[Developer IntelliJ Setup (for Kotlin)]] instructions to set it up. To build the Lingua Franca IDE (Epoch) with Kotlin-based code generators enabled (which is not possible with the Eclipse setup), please see the instructions in [[Running Lingua Franca IDE (Epoch) with Kotlin based Code Generators Enabled (without Eclipse Environment)]].

See also:

- [[Diagrams]]
- [[Regression Tests]]
- [[Legacy Eclipse Instructions]]
- [[Web Based Editor]]

Developer Eclipse setup with Oomph

Prerequisites

- Java 11 or up (download from Oracle)
- Each target language may have additional requirements. The latest release will provide the best guide.

Oomph Setup

The Eclipse setup with Oomph allows to automatically create a fully configured Eclipse IDE for the development of Lingua Franca. Note that we recommend installing a new instance of Eclipse even if you already have one for other purposes. There is no problem having multiple Eclipse installations on the same machine, and separate installations help prevent cross-project problems.

1. If you have previously installed Eclipse and you want to start fresh, then remove or move a hidden directory called `.p2` in your home directory. I do this:

```
mv ~/.p2 ~/.p2.bak
```

2. Go to the Eclipse download site (https://www.eclipse.org/downloads/index.php) and download the Eclipse Installer for your platform. The site does not advertise that it ships the Oomph Eclipse Installer but downloading Eclipse with the orange download button will give you the installer.
   You can skip this step if you already have the installer available on your system.

3. Starting the installer for the first time will open a window that looks like the following (**if you have previously followed these steps, skip to step 4**):

4. Click the Hamburger button at the top right corner and switch to "Advanced Mode".

5. Oomph now wants you to select the base Eclipse distribution for your development. We recommend to use "Eclipse IDE for Java and DSL Developers". As product version we recommend to use "Latest Release (...)". Since 2020-09, Eclipse requires Java 11+ to run.
   IMPORTANT: Xtext, used by the Lingua Franca code generator, does not yet support Java 15 or higher. Please use Java 11 through 14.
   Then press Next to continue with the project section.

6. Next, we need to register the Lingua Franca specific setup in Oomph **(only the first time you use the installer)**. Click the green Plus button at the top right corner. Select "Github

Projects" as catalog and paste the following URL into the "Resource URI" field: `https://raw.githubusercontent.com/icyphy/lingua-franca/master/oomph/LinguaFranca.setup`. Then press OK. NOTE: to check out another branch instead, adjust the URL above accordingly. For instance, in order to install the setup from `foo-bar` branch, change the URL to `https://raw.githubusercontent.com/icyphy/lingua-franca/foo-bar/oomph/LinguaFranca.setup`. Also, in the subsequent screen in the wizard, select the particular branch of interest instead of default, which is `master`.

7. Now Oomph lists the Lingua Franca setup in the "" directory of the "Github Projects" catalog. Check the Lingua Franca entry. A new entry for Lingua Franca will appear in the table at the bottom of the window. Select Lingua Franca and click Next.

8. Now you can further configure where and how your development Eclipse should be created. Check "Show all variables" to enable all possible configuration options. You can hover over the field labels to get a more detailed explanation of their effects.

- If you already have cloned the LF repository and you want Eclipse to use this location instead of cloning it into the new IDE environment, you should adjust the "Git clone location rule".
- Preferably, you have a Github account with an SSH key uploaded to Github. Otherwise, you should adjust the "Lingua Franca Github repository" entry to use the https option in the pulldown menu. See adding an SSH key to your Github account.
- If the "JRE 11 location" is empty, you need to install and/or locate a JDK that has at least version 11.

9. Click Next to get a summary of what will happen during installation. Click Finish to start.

10. Once the basic installation is complete, your new Eclipse will start. If it fails to clone the Github repository, then you should use the back button in the Oomph dialog and change the way you are accessing the repo (ssh or https). See above.
    The setup may also fail to clone the repository via SHH if Eclipse cannot find the private ssh key that matches the public key you uploaded to Github. You can configure the location of your private key in Eclipse as follows. In the Eclipse IDE, click the menu entry Window -> Preferences (on Mac Apple-Menu -> Preferences) and navigate to General -> Network Connections -> SSH2 in the tree view on the left and configure the SSH home directory and key names according to your computer. After the repo has been cloned, you can safely close the initial Oomph dialog (if not dismissed automatically). You will see a Welcome page that you can close.

11. In the new Eclipse, it may automatically start building the project, or it may pop up an "Eclipse Updater" dialog. If neither happens, you can click the button with the yellow and blue cycling arrows in the status bar at the bottom. Oomph will perform various operations to configure

the Eclipse environment, including the initial code generation for the LF language. This may take some time. Wait until the setup is finished.

12. If you get compile errors, make sure Eclipse is using Java 11. If you skipped the first step above (removing your `~/.p2` directory), then you may have legacy configuration information that causes Eclipse to mysteriously use an earlier version of Java. Lingua Franca requires Java 11 or higher, and will get compiler errors if it uses an earlier version. To fix this, go to the menu `Project->Properties` and select `Java Build Path`. Remove the entry for `JRE System Library [JRE for JavaSE-8]` (or similar). Choose `Add Library` on the right, and choose `JRE System Library`. You should now be able to choose `Workspace default JRE (JRE for JavaSE-11)`. A resulting rebuild should then compile correctly.

13. When the setup dialog is closed, your LF development IDE is ready. Probably, Eclipse is still compiling some code but when this is finished as well, all error markers on the project should have disappeared. Now, you can start a runtime Eclipse to test the actual Lingua Franca end-user IDE. In the toolbar, click on the small arrow next to the green Start button. There may already be an entry named "Launch Runtime Eclipse", but probably not. To create it, click on "Run Configurations...". Expand the "Eclipse Application" entry, select "Launch Runtime Eclipse", as follows:

Make sure that the Execution Environment shows a version of Java that is at least Java 11. The click on "Run" at the bottom.

14. A new Eclipse starts where you can write LF programs and also get a diagram representation (but you fist need to open the diagram view by clicking on Window -> Show View -> Other and selecting Diagram in the "KIELER Lightweight Diagrams" folder). You can close the welcome window in the new Eclipse and proceed to creating a new project, as below.

Using the Lingua Franca IDE

Start the Lingua Franca IDE, create a project, and create your first LF program:

- Select File->New->Project (a General Project is adequate).
- Give the project a name, like "test".
- You may want to uncheck `Use default location` and specify a location that you can remember.
- Close the Eclipse welcome window, if it is open. It obscures the project.
- Right click on the project name and select New->File.
- Give the new a name like "HelloWorld.lf" (with .lf extension).

- **IMPORTANT:** A dialog appears: Do you want to convert 'test' to an Xtext Project? Say YES.
- Start typing in Lingua-Franca! Try this:

```
target C;
main reactor HelloWorld {
    timer t;
    reaction(t) {=
        printf("Hello World.\n");
    =}
}
```

When you save, generated code goes into your project directory, e.g. `/Users/yourname/test`. That directory now has two directories inside it, `src-gen` and `bin`. The first contains the generated C code and the second contains the resulting executable program. Run the program:

```
cd ~/lingua-franca-master/runtime-EclipseXtext/test
bin/HelloWorld
```

The above directory assumes you chose default locations for everything. This should produce output that looks something like this:

```
---- Start execution at time Sun Mar 28 10:19:24 2021
---- plus 769212000 nanoseconds.
Hello World.
---- Elapsed logical time (in nsec): 0
---- Elapsed physical time (in nsec): 562,000
```

This should print "Hello World".

We strongly recommend browsing the system tests, which provide a concise overview of the capabilities of Lingua Franca. You can set up a project in the IDE for this using these instructions.

Working on the Lingua-Franca Compiler

The source code for the compiler is in the package `org.icyphy.linguafranca`.

- The grammar is in `src/org.icyphy/LinguaFranca.xtext`
- The code generator for the C target is in `src/org.icyphy.generator/CGenerator.xtend`
- The code generator for the TypeScript target is in `src/org.icyphy.generator/TypeScriptGenerator.xtend`

- The code generator for the C++ target is in `src/org.icyphy.generator/CppGenerator.xtend`
- The code generator for the Python target is in `src/org.icyphy.generator/PythonGenerator.xtend`

Troubleshooting

- GitHub uses port `443` for its ssh connections. In some systems, the default expected port by `git` can be 22, causing a timeout when cloning the repo. This can be fixed by adding the following to `~/.ssh/config`:

  ```
  Host github.com
   Hostname ssh.github.com
   Port 443
  ```

Developer IntelliJ Setup (for Kotlin)

The IntelliJ environment allows running Kotlin-based code generators (e.g., Cpp code generator), which is currently impossible for the Eclipse environment. IntelliJ also provides a debugger UI useful for debugging the code generation routines.

IntelliJ Installation

Go to the IntelliJ download page and install IntelliJ for your operating system. The community edition (CE) of IntelliJ should be enough as a development environment for Lingua Franca.

Cloning lingua-franca repository

Clone the lingua-franca repository into your working directory. For example, into a directory called `lingua-franca-intellij` using the following command:

```
$ git clone git@github.com:lf-lang/lingua-franca.git lingua-franca-intellij
```

Opening lingua-franca as IntelliJ Project

Open IntelliJ IDE. At the startup screen, click Open. Or, at the top menu bar, click File -> Open.

In the pop-up dialog, navigate to the cloned Git repository, and click Open.

If you see a pop-up saying "Maven build scripts found" after opening the repository as an IntelliJ project, click Skip since we want to use Gradle instead of Maven in this setup.

Importing Gradle Project

The next step is to load the Gradle configs for building and running Lingua Franca tasks in IntelliJ.

You might see a pop-up asking if you want to load the project as Gradle. If so, click Yes or OK.

However, in many cases, you may not see it. You can import the Gradle project by navigating to the `build.gradle` file in the root directory of the lingua-franca repository. Right-click `build.gradle`, then click `Import Gradle Project` in the drop-down list as shown below.

If you are prompted to a pop-up window asking if you trust the Gradle project, click Trust Project.

Once the repository is imported as a Gradle project, you will see a Gradle tab on the right.

You may see a message saying that Gradle project indexing is in progress at the bottom. Indexing may take a few minutes.

Once the indexing finishes, you can expand the Gradle project and see the set of Tasks.

Setting Up Run Configuration

You can set up a run configuration for running and debugging various Gradle tasks from the Gradle tab, including the code generation through LFC. To set up a run configuration for runLfc task, right-click on `runLfc` under Gradle's Tasks -> application and click `Modify Run Configuration`. This will create a custom run/debug configuration for you.

In the Run/Debug Configurations dialog, click on the text box next to Tasks: and append args to specify the LF target. For example, `runLfc --args '../example/Cpp/src/CarBrake/CarBrake.lf'` Then click OK.

You will see a new run/debug config added to the top menu bar, as shown below. You can always change the config, for example, changing the args, by clicking `Edit Configurations` via a drop-down menu.

Running and Debugging

Using the newly added config, you can run and debug the code generator by clicking the play button and the bug button.

Set up breakpoints before starting the debugger by clicking the space right next to the line numbers. While debugging, you can run code step-by-step by using the debugger tools.

Writing Reactors in C

In the C reactor target for Lingua Franca, reactions are written in C and the code generator generates a standalone C program that can be compiled and run on several platforms. It has been tested on MacOS, Linux, Windows, and at least one bare-iron embedded platforms. The single-threaded version is the most portable, requiring only a handful of common C libraries (see Included Libraries below). The multithreaded version requires a small subset of the Posix thread library (`pthreads`) and transparently executes in parallel on a multicore machine while preserving the deterministic semantics of Lingua Franca.

Note that C is not a safe language. There are many ways that a programmer can circumvent the semantics of Lingua Franca and introduce nondeterminism and illegal memory accesses. For example, it is easy for a programmer to mistakenly send a message that is a pointer to data on the stack. The destination reactors will very likely read invalid data. It is also easy to create memory leaks, where memory is allocated and never freed. Here, we provide some guidelines for a style for writing reactors that will be safe.

NOTE: On July 2, 2020, a change was pushed into the Lingua Franca repo that requires users to change their C code in reactions of reactors using the C target. Such non-backward-compatible changes are expected to be rare. If you have code that predates this change, see the summary of the changes you must make to your C code to use the new syntax. The documentation below is for the syntax required after this change.

NOTE: If you intend to use C++ code or import C++ libraries in the C target, we provide a special CCpp target that automatically uses a C++ compiler by default. Alternatively, you might want to look at the native Cpp target, built for C++ code if features such as federated execution are not needed.

A Minimal Example

A "hello world" reactor for the C target looks like this:

```
target C;
main reactor Minimal {
    reaction(startup) {=
        printf("Hello World.\n");
    =}
}
```

The `startup` trigger causes the reaction to execute at the logical start time of the program. This program can be found in a file called Minimal.lf in the test directory, where you can also find quite a

few more interesting examples. If you compile this using the `lfc` command-line compiler or the Eclipse-based IDE, then a generated file called Minimal.c plus supporting files will be put into a subdirectory called `src-gen`. In addition, the compiled C code will be put into a file called `Minimal` in a subdirectory called `bin`. If you are in the test directory, you can run this code in a shell as follows:

```
bin/Minimal
```

The resulting output should look something like this:

```
Start execution at time Sat Jun 29 11:34:41 2019
plus 958455000 nanoseconds.
Hello World.
Elapsed logical time (in nsec): 0
Elapsed physical time (in nsec): 1014000
```

It is also possible to run the executable from within the Eclipse IDE. To do this, bring up the menu on the green run button in the toolbar and select `External Tools Configurations...`. Select `Program` and click the New Configuration button. Fill out the dialog as follows (the first item is in the Variables menu and the second can be obtained by clicking on Browse Workspace, so you don't have to type them in):

You can then select the binary in the Project Explorer and, using the green Run button at the top, choose RunSelectedBinary. The output will go to the console.

The C Target Specification

To have Lingua Franca generate C code, start your `.lf` file with the following target specification:

```
target C;
```

A C target specification may optionally specify any of the parameters given in the [[target specification]]. For example, for the C target, in a source file named Foo.lf, you might specify:

```
target C {
    fast: true,
    timeout: 10 secs
};
```

The `fast` option given above specifies to execute the file as fast as possible, ignoring timing delays.

The `timeout` option specifies to stop after 10 seconds of logical time have elapsed. All events at logical time 10 seconds that have microstep 0 will be processed, but not events with later tags. If any reactor calls `request_stop` before this logical time, then the program may stop at an earlier logical time. If no timeout is specified and `request_stop` is not called, then the program will continue to run until stopped by some other mechanism (such as Control-C).

In addition, the C target supports the following parameters:

- **threads**: The number of worker threads to create to run reactions.
- **cmake**: Enable or disable the CMake-based build system (the default is `true`).
- **cmake-include**: Optionally append additional custom CMake instructions to the generated `CMakeLists.txt` from a text file.

These target specifications specify the *default* behavior of the generated code, the behavior it will exhibit if you give no command-line options. The `timeout`, `fast`, and `threads` options can be overridden on the command line when invoking the program as explained in Command-Line Arguments.

threads

```
target C {
    threads: <integer>
};
```

The number of threads to create to run reactions. This is required to be a non-negative integer. If this is not specified or a value 0 is given (the default), then a single thread is used, no thread library is included, and the `schedule()` function is not thread safe. Hence, unless you specify this, you cannot asynchronously schedule any actions. If the value given is 1 or more, then the specified number of worker threads will be created. On a multicore machine, these threads will typically be able to execute reactions in parallel if the program dependencies allow it. If a program contains physical actions, then a value of 1 or more will be forced; i.e., a threaded runtime will be used even if only one worker thread is created because the physical action needs the threading infrastructure to be able to safely asynchronous schedule events.

For example, the following

```
target C {
    threads: 4
};
```

requests that the generated code create four threads, a good choice for a four-core machine.

Important: If the `threads` option is not given, then the generated C code makes no use of a thread library. In that case, everything runs in a single thread, and asynchronous calls to `schedule()` are not supported (the `schedule()` function will not be thread safe). If the `threads` option is given, even if you request only one thread, then thread-safe code is generated and `schedule()` may be called from any thread or even from an interrupt service routine.

cmake

```
target C {
    cmake: <true or false>
};
```

Enable or disable the CMake-based build system (the default is `true`). Enabling the CMake build system will result in a `CMakeLists.txt` being generated in the `src-gen` directory. This `CMakeLists.txt` is then used when `cmake` is invoked by the LF runtime (either the `lfc` or the IDE). Alternatively, the generated program can be built manually. To do so, in the `src-gen/ProgramName` directory, run:

```
mkdir build && cd build
cmake ../
make
```

If `cmake` is disabled, `gcc` is directly invoked after code generation by default. In this case, additional target properties can be used to gain finer control over the compilation process. For example, in a source file named `Bar.lf`, you might specify:

```
target C {
    cmake: false,
    compiler: "cc",
    flags: "-g -I/usr/local/include -L/usr/local/lib -lpaho-mqtt3c"
};
```

The `compiler` option here specifies to use `cc` rather than `gcc`.

The `flags` option specifies to include debug information in the compiled code (`-g`); a directory to search for include files (`-I/usr/local/include`); a directory to search for library files (`-L/usr/local/lib`); a library to link with (`-lpaho-mqtt3c`, which will link with file `libpaho-mqtt3c.so`).

Note: Using the `flags` standard parameter when `cmake` is enabled is strongly discouraged, although supported. Flags are compiler-specific, and thus interfere with CMake's ability to find the most suitable compiler for each platform. In a similar fashion, we recommend against the use of the

`compiler` standard parameter for the same reason. A better solution is to provide a `cmake-include` file, as described next.

cmake-include

```
target C {
    cmake-include: ["relative/path/to/foo.txt", "relative/path/to/bar.txt",
};
```

Optionally append additional custom CMake instructions to the generated `CMakeLists.txt` from text files (e.g, `foo.txt`). The specified files are resolved using the file search algorithm documented [[here | Target-Specification#files]], and copied to the directory that contains the generated sources (only in the C target). This is done to make the generated code more portable (a feature that is useful in [[federated execution | Distributed-Execution]]).

This target property can be used, for example, to add dependencies for various packages (e.g., by using `find_package()` and `target_link_libraries` commands). CMakeInclude.lf is an example that uses this feature. A more sophisticated example of the usage of this target parameter can be found in Rhythm.lf.

A CMake variable called `${LF_MAIN_TARGET}` can be used in the included text file(s) for convenience. This variable will contain the name of the CMake target (i.e., the name of the main reactor). For example, a `foo.txt` file can contain:

```
find_package(m REQUIRED) # Finds the m library

target_link_libraries( ${LF_MAIN_TARGET} m ) # Links the m library
```

`foo.txt` can then be included:

```
target C {
    cmake-include: "foo.txt"
};
```

In this case, "foo.txt" is in the same `src` folder as the main `.lf` file.

Note: For a general tutorial on finding packages in CMake, see this external documentation entry. For a list of CMake find modules, see this.

Finally, `cmake-include` works in conjunction with import. If any imported `.lf` file has `cmake-include` in its target property, it will be appended to the current list of `cmake-include`s. These files will be resolved relative to the imported `.lf` file using the [[files | Target-Specification#files]] search procedure and copied to the directory that contains the generated sources. This will help

resolve dependencies in imported reactors automatically and make the code more portable. DistributedCMakeInclude.lf is a test that uses this feature.

Note: For [[federated execution | Distributed-Execution]], both `cmake-include` and `file` are kept separate for each federate as much as possible. This means that if one federate is imported, or uses an imported reactor that other federates don't use, it will only have access to `cmake-include`s and `file`s defined in the main `.lf` file, plus the selectively imported `.lf` files. DistributedCMakeIncludeSeparateCompile.lf is a test that demonstrates this feature.

Command-Line Arguments

The generated C program understands the following command-line arguments, each of which has a short form (one character) and a long form:

- `-f, --fast [true | false]`: Specifies whether to wait for physical time to match logical time. The default is `false`. If this is `true`, then the program will execute as fast as possible, letting logical time advance faster than physical time.
- `-o, --timeout <duration> <units>`: Stop execution when logical time has advanced by the specified *duration*. The units can be any of nsec, usec, msec, sec, minute, hour, day, week, or the plurals of those.
- `-k, --keepalive [true | false]`: Specifies whether to stop execution if there are no events to process. This defaults to `false`, meaning that the program will stop executing when there are no more events on the event queue. If you set this to `true`, then the program will keep executing until either the `timeout` logical time is reached or the program is externally killed. If you have `physical action`s, it usually makes sense to set this to `true`.
- `-t, --threads <n>`: Executed in threads if possible. This option is ignored in the single-threaded version. That is, it is ignored if a `threads` option was not given in the source .lf file.

Any other command-line arguments result in printing the above information.

Imports

The import statement can be used to share reactor definitions across several applications. Suppose for example that we modify the above Minimal.lf program as follows and store this in a file called HelloWorld.lf:

```
target C;
reactor HelloWorld {
    reaction(startup) {=
        printf("Hello World.\n");
    =}
}
main reactor HelloWorldTest {
    a = new HelloWorld();
}
```

This can be compiled and run, and its behavior will be identical to the version above. But now, this can be imported into another reactor definition as follows:

```
target C;
import HelloWorld.lf;
main reactor TwoHelloWorlds {
    a = new HelloWorld();
    b = new HelloWorld();
}
```

This will create two instances of the HelloWorld reactor, and when executed, will print "Hello World" twice.

Note that in the above example, the order in which the two reactions are invoked is undefined because there is no causal relationship between them. In fact, if you modify the target specification to say:

```
target C {threads: 2};
```

then you might see garbled output if the implementation of `printf` on your machine is not thread safe (most modern implementations *are* thread safe, so you are not likely to see this behavior).

A more interesting illustration of imports can be found in the Import.lf test case in the test directory.

Preamble

Reactions may contain arbitrary C code, but often it is convenient for that code to invoke external libraries or to share procedure definitions. For either purpose, a reactor may include a **preamble** section. For example, the following reactor uses the common `stdlib` C library to convert a string to an integer:

```
main reactor Preamble {
    preamble {=
        #include <stdlib.h>
    =}
    timer t;
    reaction(t) {=
        char* s = "42";
        int i = atoi(s);
        printf("Converted string %s to int %d.\n", s, i);
    =}
}
```

This will print:

```
Converted string 42 to int 42.
```

By putting the `#include` in the **preamble**, the library becomes available in all reactions of this reactor. Oddly, it also becomes available in all subsequently defined reactors in the same file or in files that include this file.

You can also use the preamble to define functions that are shared across reactions and reactors (this is Preamble.lf in the test suite):

```
main reactor Preamble {
    preamble {=
        int add_42(int i) {
            return i + 42;
        }
    =}
    timer t;
    reaction(t) {=
        printf("42 plus 42 is %d.\n", add_42(42));
    =}
}
```

Not surprisingly, this will print:

```
42 plus 42 is 84.
```

Reactions

Recall that a reaction is defined within a reactor using the following syntax:

reaction(*triggers*) *uses* -> *effects* {=
  ... target language code ...
=}

In this section, we explain how **triggers**, **uses**, and **effects** variables work in the C target.

Inputs and Outputs

In the body of a reaction in the C target, the value of an input is obtained using the syntax `name->value`, where `name` is the name of the input port. To determine whether an input is present, use `name->is_present`. For example, the Determinism.lf test case in the test directory includes the following reactor:

```
reactor Destination {
    input x:int;
    input y:int;
    reaction(x, y) {=
        int sum = 0;
        if (x->is_present) {
            sum += x->value;
        }
        if (y->is_present) {
            sum += y->value;
        }
        printf("Received %d.\n", sum);
    =}
}
```

The reaction refers to the input values `x->value` and `y->value` and tests for their presence by referring to the variables `x->is_present` and `y->is_present`. If a reaction is triggered by just one input, then normally it is not necessary to test for its presence; it will always be present. But in the above example, there are two triggers, so the reaction has no assurance that both will be present.

Inputs declared in the **uses** part of the reaction do not trigger the reaction. Consider this modification of the above reaction:

```
reaction(x) y {=
    int sum = x->value;
    if (y->is_present) {
        sum += y->value;
    }
    printf("Received %d.\n", sum);
=}
```

It is no longer necessary to test for the presence of `x` because that is the only trigger. The input `y`, however, may or may not be present at the logical time that this reaction is triggered. Hence, the code must test for its presence.

The **effects** portion of the reaction specification can include outputs and actions. Actions will be described below. Outputs are set using a `SET` macro. For example, we can further modify the above example as follows:

```
output z:int;
reaction(x) y -> z {=
    int sum = x->value;
    if (y->is_present) {
        sum += y->value;
    }
    SET(z, sum);
=}
```

The `SET` macro is shorthand for this:

```
z->value = sum;
z->is_present = true;
```

There are several variants of the `SET` macro, and the one you should use depends on the type of the output. The simple version shown above works for all primitive C type (int, double, etc.) as well as the `bool` and `string` types that Lingua Franca defines. For the other variants, see Sending and Receiving Arrays and Structs below.

If an output gets set more than once at any logical time, downstream reactors will see only the *final* value that is set. Since the order in which reactions of a reactor are invoked at a logical time is deterministic, and whether inputs are present depends only on their timestamps, the final value set for an output will also be deterministic.

An output may even be set in different reactions of the same reactor at the same logical time. In this case, one reaction may wish to test whether the previously invoked reaction has set the output. It can check `name->is_present` to determine whether the output has been set. For example, the following reactor (see TestForPreviousOutput.lf) will always produce the output 42:

```
reactor TestForPreviousOutput {
    output out:int;
    reaction(startup) -> out {=
        // Set a seed for random number generation based on the current tim
        srand(time(0));
        // Randomly produce an output or not.
        if (rand() % 2) {
            SET(out, 21);
        }
    =}
    reaction(startup) -> out {=
        if (out->is_present) {
            SET(out, 2 * out->value);
        } else {
            SET(out, 42);
        }
    =}
}
```

The first reaction may or may not set the output to 21. The second reaction doubles the output if it has been previously produced and otherwise produces 42.

Using State Variables

A reactor may declare state variables, which become properties of each instance of the reactor. For example, the following reactor (see Count.lf and CountTest.lf) will produce the output sequence 1, 2, 3, ... :

```
reactor Count {
    state count:int(1);
    output y:int;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        SET(y, self->count++);
    =}
}
```

The declaration on the second line gives the variable the name "count", declares its type to be `int`, and initializes its value to 1. The type and initial value can be enclosed in the C-code delimiters `{= ... =}` if they are not simple identifiers, but in this case, that is not necessary.

NOTE: String types in C are `char*`. But, as explained below, types ending with `*` are interpreted specially to provide automatic memory management, which we generally don't want with strings (a

string that is a compile-time constant must not be freed). You could enclose the type as `{= char* =}`, but to avoid this awkwardness, the header files include a typedef that permits using `string` instead of `char*`. For an example using `string` data types, see DelayString.lf.

In the body of the reaction, the state variable is referenced using the syntax `self->count`. Here, **self** is a keyword that is provided by Lingua Franca. It refers to a struct that contains all the instance-specific data associated with an instance of the reactor. Since each instance of a reactor has its own state variables, these variables are carried in the **self** struct.

It may be tempting to declare state variables in the **preamble**, as follows:

```
reactor FlawedCount {
    preamble {=
        int count = 0;
    =}
    output y:int;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        SET(y, count++);
    =}
}
```

This will produce a sequence of integers, but if there is more than one instance of the reactor, those instances will share the same variable count. Hence, **don't do this**! Sharing variables across instances of reactors violates a basic principle, which is that reactors communicate only by sending messages to one another. Sharing variables will make your program nondeterministic. If you have multiple instances of the above FlawedCount reactor, the outputs produced by each instance will not be predictable, and in a multithreaded implementation, will also not be repeatable.

A state variable may be a time value, declared as follows (see for example SlowingClock.lf):

```
    state time_value:time(100 msec);
```

The `self->time_value` variable will be of type `instant_t`, which is a `long long` and the same type as `interval_t`. The value of the variable is a number in units of nanoseconds.

A state variable can have an array value. For example, the [MovingAverage] (https://github.com/lf-lang/lingua-franca/blob/master/test/C/src/MovingAverage.lf) reactor computes the **moving average** of the last four inputs each time it receives an input:

```
reactor MovingAverageImpl {
    state delay_line:double[](0.0, 0.0, 0.0);
    state index:int(0);
    input in:double;
    output out:double;
    reaction(in) -> out {=
        // Calculate the output.
        double sum = in->value;
        for (int i = 0; i < 3; i++) {
            sum += self->delay_line[i];
        }
        SET(out, sum/4.0);

        // Insert the input in the delay line.
        self->delay_line[self->index] = in->value;

        // Update the index for the next input.
        self->index++;
        if (self->index >= 3) {
            self->index = 0;
        }
    =}
}
```

The second line declares that the type of the state variable is an array of `double`s with the initial value of the array being a three-element array filled with zeros.

States whose type are structs can similarly be initialized. See StructAsState.lf.

Using Parameters

Reactor parameters are also referenced in the C code using the **self** struct. The Stride example modifies the above `Count` reactor so that its stride is a parameter:

```
target C;
reactor Count(stride:int(1)) {
    state count:int(1);
    output y:int;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        SET(y, self->count);
        self->count += self->stride;
    =}
}
reactor Display {
    input x:int;
    reaction(x) {=
        printf("Received: %d.\n", x->value);
    =}
}
main reactor Stride {
    c = new Count(stride = 2);
    d = new Display();
    c.y -> d.x;
}
```

The second line defines the `stride` parameter, gives its type, and gives its initial value. As with state variables, the type and initial value can be enclosed in `{= ... =}` if necessary. The parameter is referenced in the reaction with the syntax `self->stride`.

When the reactor is instantiated, the default parameter value can be overridden. This is done in the above example near the bottom with the line:

```
    c = new Count(stride = 2);
```

If there is more than one parameter, use a comma separated list of assignments.

Parameters can have array values, though some care is needed. The ArrayAsParameter example outputs the elements of an array as a sequence of individual messages:

```
reactor Source(sequence:int[](0, 1, 2), n_sequence:int(3)) {
    output out:int;
    state count:int(0);
    logical action next;
    reaction(startup, next) -> out, next {=
        SET(out, self->sequence[self->count]);
        self->count++;
        if (self->count < self->n_sequence) {
            schedule(next, 0);
        }
    =}
}
```

The **logical action** named `next` and the `schedule` function are explained below in Scheduling Delayed Reactions; here they are used simply to repeat the reaction until all elements of the array have been sent.

In C, arrays do not encode their own length, so a separate parameter is used for the array length. Obviously, there is potential here for errors, where the array length doesn't match the length parameter.

Above, the parameter default value is an array with three elements, `[0, 1, 2]`. The syntax for giving this default value is that of a Lingua Franca list, `(0, 1, 2)`, which gets converted by the code generator into a C static initializer. The default value can be overridden when instantiating the reactor using a similar syntax:

```
    s = new Source(sequence = (1, 2, 3, 4), n_sequence=4);
```

Sending and Receiving Arrays and Structs

You can define your own datatypes in C and send and receive those. Consider the StructAsType example:

```
reactor StructAsType {
    preamble {=
        typedef struct hello_t {
            char* name;
            int value;
        } hello_t;
    =}
    output out:hello_t;
    reaction(startup) -> out {=
        struct hello_t temp = {"Earth", 42};
        SET(out, temp);
    =}
}
```

The **preamble** code defines a struct datatype. In the reaction to **startup**, the reactor creates an instance of this struct on the stack (as a local variable named `temp`) and then copies that struct to the output using the `SET` macro.

For large structs, it may be inefficient to create a struct on the stack and copy it to the output, as done above. You can instead write directly to the fields of the struct. For example, the above reaction could be rewritten as follows (see StructAsTypeDirect):

```
    reaction(startup) -> out {=
        out->value.name = "Earth";
        out->value.value = 42;
        SET_PRESENT(out);
    =}
```

The final call to `SET_PRESENT` is necessary to inform downstream reactors that the struct has a new value. (This is a macro that simply does `out->is_present = true`). Note that in subsequent reactions, the values of the struct persist. Hence, this technique can be very efficient if a large struct is modified only slightly in each of a sequence of reactions.

A reactor receiving the struct message uses the struct as normal in C:

```
reactor Print() {
    input in:hello_t;
    reaction(in) {=
        printf("Received: name = %s, value = %d\n", in->value.name, in->val
    =}
}
```

The preamble should not be repeated in this reactor definition if the two reactors are defined together because this will trigger an error when the compiler thinks that hello_t is being redefined.

Arrays that have fixed sizes are handled similarly. Consider the ArrayAsType example:

```
reactor ArrayAsType {
    output out:int[3];
    reaction(startup) -> out {=
        out[0] = 0;
        out[1] = 1;
        out[2] = 2;
        SET_PRESENT(out);
    =}
}
```

Here, the output is declared to have type `int[3]`, an array of three integers. The startup reaction above writes to the array and then calls `SET_PRESENT` to indicate an updated value. Again, the values in the array will persist across reactions.

A reactor receiving this array is straightforward. It just references the array elements as usual in C, as illustrated by this example:

```
reactor Print() {
    input in:int[3];
    reaction(in) {=
        printf("Received: [");
        for (int i = 0; i < 3; i++) {
            if (i > 0) printf(", ");
            printf("%d", in->value[i]);
        }
        printf("]\n");
    =}
}
```

Dynamically Allocated Arrays

For arrays where the size is variable, it may be necessary to dynamically allocate memory. But when should that memory be freed? A reactor cannot know when downstream reactors are done with the data. Lingua Franca provides utilities for managing this using reference counting. You can pass a pointer to a dynamically allocated object as illustrated in the ArrayPrint example:

```
reactor ArrayPrint {
    output out:int[];
    reaction(startup) -> out {=
        // Dynamically allocate an output array of length 3.
        SET_NEW_ARRAY(out, 3);
        // Above allocates the array, which then must be populated.
        out[0] = 0;
        out[1] = 1;
        out[2] = 2;
    =}
}
```

This declares the output datatype to be `int[]` (or, equivalently, `int*`), an array of integers of unspecified size. To produce the array in a reaction, it uses the library function `SET_NEW_ARRAY` to allocate an array of length 3 and sets the output to send that array. The reaction then populates the array with data. The deallocation of the memory for the array will occur automatically after the last reactor that receives a pointer to the array has finished using it.

A reactor receiving the array looks like this:

```
reactor Print {
    input in:int[];
    reaction(in) {=
        printf("Received: [");
        for (int i = 0; i < in->length; i++) {
            if (i > 0) printf(", ");
            printf("%d", in->value[i]);
        }
        printf("]\n");
    =}
}
```

In the body of the reaction, `in->value` is a pointer to first element of the array, so it can be indexed as usual with arrays in C, `in->value[i]`. Moreover, a variable `in->length` is bound to the length of the array.

Although it cannot be enforced in C, the receiving reactor should not modify the values stored in the array. Inputs are logically *immutable* because there may be several recipients. Any recipient that wishes to modify the array should make a copy of it. Fortunately, a utility is provided for this pattern. Consider the ArrayScale example:

```
reactor ArrayScale(scale:int(2)) {
    mutable input in:int[];
    output out:int[];
    reaction(in) -> out {=
        for(int i = 0; i < in->length; i++) {
            in->value[i] *= self->scale;
        }
        SET_TOKEN(out, in->token);
    =}
}
```

Here, the input is declared **mutable**, which means that any reaction is free to modify the input. If this reactor is the only recipient of the array or the last recipient of the array, then this will not copy of the array but rather use the original array. Otherwise, it will use a copy.

The above `ArrayScale` reactor modifies the array and then forwards it to its output port using the `SET_TOKEN()` macro. That macro further delegates to downstream reactors the responsibility for freeing dynamically allocated memory once all readers have completed their work.

If the above code were not to forward the array, then the dynamically allocated memory will be automatically freed when this reactor is done with it.

The above three reactors can be combined into a pipeline as follows:

```
main reactor ArrayScaleTest {
    s = new ArrayPrint();
    c = new ArrayScale();
    p = new Print();
    s.out -> c.in;
    c.out -> p.in;
}
```

In this composite, the array is allocated by `ArrayPrint`, modified by `ArrayScale`, and deallocated (freed) after `Print` has reacted. No copy is necessary because `ArrayScale` is the only recipient of the original array.

Inputs and outputs can also be dynamically allocated structs. In fact, Lingua Franca's C target will treat any input or output datatype that ends with `[]` or `*` specially by providing utilities for allocating memory and modifying and forwarding. Deallocation of the allocated memory is automatic. The complete set of utilities is given below.

Macros For Setting Output Values

In all of the following, *out* is the name of the output and *value* is the value to be sent.

SET(*out*, *value*); Set the specified output (or input of a contained reactor) to the specified value. This version is used for primitive type such as `int`, `double`, etc. as well as the built-in types `bool` and `string` (but only if the string is a statically allocated constant; otherwise, see `SET_NEW_ARRAY`). It can also be used for structs with a type defined by a `typedef` so that the type designating string does not end in '*'. The value is copied and therefore the variable carrying the value can be subsequently modified without changing the output.

SET_ARRAY(*out*, *value*, *element_size*, *length*); This version is used for outputs with a type declaration ending with `[]` or `*`, such as `int[]`. This version is for use when the *value* to be sent is in dynamically allocated memory that will need to be freed downstream. The allocated memory will be automatically freed when all recipients of the outputs are done with it. Since C does not encode array sizes as part of the array, the *length* and *element_size* must be given (the latter is the size of each element in bytes). See SetArray.lf.

SET_NEW(*out*); This version is used for outputs with a type declaration ending with `*` (see example below). This sets the `out` variable to point to newly allocated memory for storing the specified output type. After calling this function, the reaction should populate that memory with the content it intends to send to downstream reactors. This macro is equivalent to `SET_NEW_ARRAY(out, 1)`. See StructPrint.lf

SET_NEW_ARRAY(*out*, *length*); This version is used for outputs with a type declaration ending with `[]` or `*`. This sets the *out* variable to point to newly allocated memory sufficient to hold an array of the specified length containing the output type in each element. The caller should subsequently populate the array with the contents that it intends to send to downstream reactors. See ArrayPrint.lf. **Dynamically allocated strings:** If an output is to be a dynamically allocated string, as opposed to a static string constant, then you can use `SET_NEW_ARRAY` to allocate the memory, and the memory will be automatically freed downstream after the all users have read the string. To do this, set the output type to `char[]` or `char*` rather than `string` and call `SET_NEW_ARRAY` with the desired length. After this, *out* will point to a char array of the required length. You can then populate it with your desired string, e.g. using `snprintf()`. See DistributedToken.lf

SET_PRESENT(*out*); This version just sets the *out*->is_present variable corresponding to the specified output to true. This is normally used with array outputs with fixed sizes and statically allocated structs. In these cases, the values in the output are normally written directly to the array or struct. See ArrayAsType.lf

SET_TOKEN(*out*, *value*); This version is used for outputs with a type declaration ending with `*` (any pointer) or `[]` (any array). The *value* argument should be a struct of type `token_t`. This can be the trickiest form to use, but it is rarely necessary for the programmer to create their own (dynamically allocated) instance of `token_t`. Consider the SetToken.lf example:

```
reactor Source {
    output out:int*;
    logical action a:int;
    reaction(startup) -> a {=
        schedule_int(a, MSEC(200), 42);
    =}
    reaction(a) -> out {=
        SET_TOKEN(out, a->token);
    =}
}
```

Here, the first reaction schedules an integer-valued action to trigger after 200 microseconds. As explained below, action payloads are carried by tokens. The second reaction grabs the token rather than the value using the syntax `a->token` (the name of the action followed by `->token`). It then forwards the token to the output. The output data type is `int*` not `int` because the token carries a pointer to dynamically allocated memory that contains the value. All inputs and outputs with types ending in `*` or `[]` are carried by tokens.

All of the SET macros will overwrite any output value previously set at the same logical time and will cause the final output value to be sent to all reactors connected to the output. They also all set a local *out*->is_present variable to true. This can be used to subsequently test whether the output value has been set.

Dynamically Allocated Structs

The `SET_NEW` and `SET_TOKEN` macros can be used to send `structs` of arbitrary complexity. For example:

```
reactor StructPrint {
    preamble {=
        typedef struct hello_t {
            char* name;
            int value;
        } hello_t;
    =}
    output out:hello_t*;
    reaction(startup) -> out {=
        // Dynamically allocate an output struct.
        SET_NEW(out);
        // Above allocates a struct, which then must be populated.
        out->value->name = "Earth";
        out->value->value = 42;
    =}
}
```

The **preamble** declares a struct type `hello_t` with two fields, and the `SET_NEW` macro allocates memory to contain such a struct. The subsequent code populates that memory. A reactor receiving this struct might look like this:

```
reactor Print() {
    input in:hello_t*;
    reaction(in) {=
        printf("Received: name = %s, value = %d\n",
            in->value->name, in->value->value
        );
    =}
}
```

Just as with arrays, an input with a pointer type can be declared **mutable**, in which case it is safe to modify the fields and forward the struct.

Occasionally, you will want an input or output type to be a pointer, but you don't want the automatic memory allocation and deallocation. A simple example is a string type, which in C is `char*`. Consider the following (erroneous) reactor:

```
reactor Erroneous {
    output out:char*;
    reaction(startup) -> out {=
        SET(out, "Hello World");
    =}
}
```

An output data type that ends with `*` signals to Lingua Franca that the message is dynamically allocated and must be freed downstream after all recipients are done with it. But the "Hello World" string here is statically allocated, so an error will occur when the last downstream reactor to use this message attempts to free the allocated memory. To avoid this for strings, you can use the `string` type, defined in `reactor.h`, as follows:

```
reactor Fixed {
    output out:string;
    reaction(startup) -> out {=
        SET(out, "Hello World");
    =}
}
```

The `string` type is equivalent to `char*`, but since it doesn't end with `*`, it does not signal to Lingua Franca that the type is dynamically allocated. Lingua Franca only handles allocation and deallocation for types that are specified literally with a final `*` in the type name. The same trick can be used for any type where you don't want automatic allocation and deallocation. E.g., the SendsPointer example looks like this:

```
reactor SendsPointer  {
    preamble {=
        typedef int* int_pointer;
    =}
    output out:int_pointer;
    reaction(startup) -> out {=
        static int my_constant = 42;
        SET(out, &my_constant;)
    =}
}
```

The above technique can be used to abuse the reactor model of computation by communicating pointers to shared variables. This is generally a bad idea unless those shared variables are immutable. The result will likely be nondeterministic. Also, communicating pointers across machines that do not share memory will not work at all.

Timed Behavior

Timers are specified exactly as in the Lingua Franca language specification. When working with time in the C code body of a reaction, however, you will need to know a bit about its internal representation.

In the C target, the value of a time instant or interval is an integer specifying a number of nanoseconds. An instant is the number of nanoseconds that have elapsed since January 1, 1970. An interval is the difference between two instants. When an LF program starts executing, logical time is (normally) set to the instant provided by the operating system. (On some embedded platforms without real-time clocks, it will be set instead to zero.)

Time in the C target is a `long long`, which is (normally) a 64 bit signed number. Since a 64-bit number has a limited range, this measure of time instants will overflow in approximately the year 2262. For better code clarity, two types are defined in tag.h, `instant_t` and `interval_t`, which you can use for time instants and intervals respectively. These are both equivalent to `long long`, but using those types will insulate your code against changes and platform-specific customizations.

Lingua Franca uses a superdense model of time. A reaction is invoked at a logical **tag**, a struct consists of a `time` value (an `instant_t`, which is a `long long`) and a `microstep` value (a `microstep_t`, which is an unsigned `int`). The tag is guaranteed to not increase during the execution of a reaction. Outputs produced by a reaction have the same tag as the inputs, actions, or timers that trigger the reaction, and hence are **logically simultaneous**.

The time structs and functions for working with time are defined in tag.h. The most useful functions are:

- `tag_t get_current_tag()`: Get the current tag at which this reaction has been invoked.
- `instant_t get_logical_time()`: Get the current logical time (the first part of the current tag).
- `microstep_t get_microstep()`: Get the current microstep (the second part of the current tag).
- `interval_t get_elapsed_logical_time()`: Get the logical time elapsed since program start.
- `int compare_tags(tag_t, tag_t)`: Compare two tags, returning -1, 0, or 1 for less than, equal, and greater than.

There are also some useful functions for accessing physical time:

- `instant_t get_physical_time()`: Get the current physical time.
- `instant_t get_elapsed_physical_time()`: Get the physical time elapsed since program start.
- `instant_t get_start_time()`: Get the starting physical and logical time.

The last of these is both a physical and logical time because, at the start of execution, the starting logical time is set equal to the current physical time as measured by a local clock.

A reaction can examine the current logical time (which is constant during the execution of the reaction). For example, consider the GetTime example:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        instant_t logical = get_logical_time();
        printf("Logical time is %lld.\n", logical);
    =}
}
```

When executed, you will get something like this:

```
Start execution at time Sun Oct 13 10:18:36 2019
plus 353609000 nanoseconds.
Logical time is 1570987116353609000.
Logical time is 1570987117353609000.
Logical time is 1570987118353609000.
...
```

The first two lines give the current time-of-day provided by the execution platform at the start of execution. This is used to initialize logical time. Subsequent values of logical time are printed out in their raw form, rather than the friendlier form in the first two lines. If you look closely, you will see that each number is one second larger than the previous number, where one second is 1000000000 nanoseconds.

You can also obtain the *elapsed* logical time since the start of execution:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        interval_t elapsed = get_elapsed_logical_time();
        printf("Elapsed logical time is %lld.\n", elapsed);
    =}
}
```

This will produce:

```
Start execution at time Sun Oct 13 10:25:22 2019
plus 833273000 nanoseconds.
Elapsed logical time is 0.
Elapsed logical time is 1000000000.
Elapsed logical time is 2000000000.
...
```

You can also get physical time, which comes from your platform's real-time clock:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        instant_t physical = get_physical_time();
        printf("Physical time is %lld.\n", physical);
    =}
}
```

This will produce something like this:

```
Start execution at time Sun Oct 13 10:35:59 2019
plus 984992000 nanoseconds.
Physical time is 1570988159986108000.
Physical time is 1570988160990219000.
Physical time is 1570988161990067000.
...
```

Finally, you can get elapsed physical time:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        instant_t elapsed_physical = get_elapsed_physical_time();
        printf("Elapsed physical time is %lld.\n", elapsed_physical);
    =}
}
```

This will produce something like this:

```
Elapsed physical time is 657000.
Elapsed physical time is 1001856000.
Elapsed physical time is 2004761000.
...
```

Notice that these numbers are increasing by *roughly* one second each time. If you set the `fast` target parameter to `true`, then physical time will elapse much faster than logical time.

Working with nanoseconds in C code can be tedious if you are interested in longer durations. For convenience, a set of macros are available to the C programmer to convert time units into the required nanoseconds. For example, you can specify 200 msec in C code as `MSEC(200)` or two weeks as `WEEKS(2)`. The provided macros are NSEC, USEC (for microseconds), MSEC, SEC,

MINUTE, HOUR, DAY, and WEEK. You may also use the plural of any of these. Examples are given in the next section.

Scheduling Delayed Reactions

The C target provides a variety of `schedule()` functions to trigger an action at a future logical time. Actions are described in the Language Specification document. Consider the Schedule reactor:

```
target C;
reactor Schedule {
    input x:int;
    logical action a;
    reaction(a) {=
        interval_t elapsed_time = get_elapsed_logical_time();
        printf("Action triggered at logical time %lld nsec after start.\n",
    =}
    reaction(x) -> a {=
        schedule(a, MSEC(200));
    =}
}
```

When this reactor receives an input `x`, it calls `schedule()`, specifying the action `a` to be triggered and the logical time offset (200 msec). The action `a` will be triggered at a logical time 200 milliseconds after the arrival of input `x`. At that logical time, the second reaction will trigger and will use the `get_elapsed_logical_time()` function to determine how much logical time has elapsed since the start of execution.

Notice that after the logical time offset of 200 msec, there may be another input `x` simultaneous with the action `a`. Because the reaction to `a` is given first, it will execute first. This becomes important when such a reactor is put into a feedback loop (see below).

Zero-Delay Actions

If the specified delay in a `schedule()` call is zero, then the action `a` will be triggered one **microstep** later in **superdense time** (see Superdense Time). Hence, if the input `x` arrives at metric logical time *t*, and you call `schedule()` as follows:

```
schedule(a, 0);
```

then when a reaction to `a` is triggered, the input `x` will be absent (it was present at the *previous* microstep). The reaction to `x` and the reaction to `a` occur at the same metric time *t*, but separated by one microstep, so these two reactions are *not* logically simultaneous.

The metric time is visible to the C programmer and can be obtained in a reaction using either `get_elapsed_logical_time()`, as above, or `get_logical_time()`. The latter function also returns a `long long` (aka `instant_t`), but its meaning is now the time elapsed since January 1, 1970 in nanoseconds.

As described in the Language Specification document, action declarations can have a *min_delay* parameter. This modifies the timestamp further. Also, the action declaration may be **physical** rather than **logical**, in which case, the assigned timestamp will depend on the physical clock of the executing platform.

Actions With Values

If an action is declared with a data type, then it can carry a **value**, a data value that becomes available to any reaction triggered by the action. This is particularly useful for physical actions that are externally triggered because it enables the action to convey information to the reactor. This could be, for example, the body of an incoming network message or a numerical reading from a sensor.

Recall from the Contained Reactors section in the Language Specification document that the **after** keyword on a connection between ports introduces a logical delay. This is actually implemented using a logical action. We illustrate how this is done using the DelayInt example:

```
reactor DelayInt(delay:time(100 msec)) {
    input in:int;
    output out:int;
    logical action d:int;
    reaction(d) -> out {=
        SET(out, d->value);
    =}
    reaction(in) -> d {=
        schedule_int(d, self->delay, in->value);
    =}
}
```

Using this reactor as follows

```
    d = new Delay();
    source.out -> d.in;
    d.in -> sink.out
```

is equivalent to

```
    source.out -> sink.in after 100 msec
```

(except that our `DelayInt` reactor will only work with data type `int`).

The action `d` is specified with a type `int`. The reaction to the input `in` declares as its effect the action `d`. This declaration makes it possible for the reaction to schedule a future triggering of `d`. The reaction uses one of several variants of the **schedule** function, namely **schedule_int**, a convenience function provided because integer payloads on actions are very common. We will see below, however, that payloads can have any data type.

The first reaction declares that it is triggered by `d` and has effect `out`. To read the value, it uses the `d->value` variable. Because this reaction is first, the `out` at any logical time can be produced before the input `in` is even known to be present. Hence, this reactor can be used in a feedback loop, where `out` triggers a downstream reactor to send a message back to `in` of this same reactor. If the reactions were given in the opposite order, there would be causality loop and compilation would fail.

If you are not sure whether an action carries a value, you can test for it as follows:

```
    reaction(d) -> out {=
        if (d->has_value) {
            SET(out, d->value);
        }
    =}
```

It is possible to both be triggered by and schedule an action the same reaction. For example, this CountSelf reactor will produce a counting sequence after it is triggered the first time:

```
reactor CountSelf(delay:time(100 msec)) {
    output out:int;
    logical action a:int;
    reaction(startup) -> a, out {=
        SET(out, 0);
        schedule_int(a, self->delay, 1);
    =}
    reaction(a) -> a, out {=
        SET(out, a->value);
        schedule_int(a, self->delay, a->value + 1);
    =}
}
```

Of course, to produce a counting sequence, it would be more efficient to use a state variable.

Actions with values can be rather tricky to use because the value must usually be carried in dynamically allocated memory. It will not work for value to refer to a state variable of the reactor because that state variable will likely have changed value by the time the reactions to the action are

invoked. Several variants of the **schedule** function are provided to make it easier to pass values across time in varying circumstances.

> **schedule**(*action*, *offset*); This is the simplest version as it carries no value. The action need not have a data type.
>
> **schedule_int**(*action*, *offset*, *value*); This version carries an `int` value. The datatype of the action is required to be `int`.
>
> **schedule_token**(*action*, *offset*, *value*); This version carries a **token**, which has type `token_t` and points to the value, which can have any type. There is a `create_token()` function that can be used to create a token, but programmers will rarely need to use this. Instead, you can use `schedule_value()` (see below), which will automatically create a token. Alternatively, for inputs with types ending in `*` or `[]`, the value is wrapped in a token, and the token can be obtained using the syntax `inputname->token` in a reaction and then forwarded using `schedule_token()` (see section Dynamically Allocated Structs above). If the input is mutable, the reaction can then even modify the value pointed to by the token and/or use `schedule_token()` to send the token to a future logical time. For example, the DelayPointer reactor realizes a logical delay for any datatype carried by a token:

```
reactor DelayPointer(delay:time(100 msec)) {
    input in:void*;
    output out:void*;
    logical action a:void*;
    reaction(a) -> out {=
        // Using SET_TOKEN delegates responsibility for
        // freeing the allocated memory downstream.
        SET_TOKEN(out, a->token);
    =}
    reaction(in) -> a {=
        // Schedule the actual token from the input rather than
        // a new token with a copy of the input value.
        schedule_token(a, self->delay, in->token);
    =}
}
```

> **schedule_value**(*action*, *offset*, *value*, *length*); This version is used to send into the future a value that has been dynamically allocated malloc. It will be automatically freed when it is no longer needed. The *value* argument is a pointer to the memory containing the value. The *length* argument should be 1 if it is a not an array and the array length otherwise. This length will be needed downstream to interpret the data correctly. See ScheduleValue.lf.

> **schedule_copy**(*action*, *offset*, *value*, *length*); This version is for sending a copy of some data pointed to by the *value* argument. The data is assumed to be a scalar or array of type matching the *action* type. The *length* argument should be 1 if it is a not an array and the array length otherwise. This length will be needed downstream to interpret the data correctly.

Occasionally, an action payload may not be dynamically allocated nor freed. For example, it could be a pointer to a statically allocated string. If you know this to be the case, the DelayString reactor will realize a logical time delay on such a string:

```
reactor DelayString(delay:time(100 msec)) {
    input in:string;
    output out:string;
    logical action a:string;
    reaction(a) -> out {=
        SET(out, a->value);
    =}
    reaction(in) -> a {=
        // The following copies the char*, not the string.
        schedule_copy(a, self->delay, &(in->value), 1);
    =}
}
```

The datatype `string` is an alias for `char*`, but Lingua Franca does not know this, so it creates a token that contains a copy of the pointer to the string rather than a copy of the string itself.

Stopping Execution

A reaction may request that the execution stop after all events with the current timestamp have been processed by calling the built-in function `request_stop()`, which takes no arguments. In a non-federated execution, the returned time is normally the same as the current logical time, and the actual last tag will be one microstep later. In a federated execution, however, the stop time will likely be larger than the current logical time. All federates are assured of stopping at the same logical time.

Log and Debug Information

A suite of useful functions is provided in util.h for producing messages to be made visible when the generated program is run. Of course, you can always use `printf`, but this is not a good choice for logging or debug information, and it is not a good choice when output needs to be redirected to a window or some other user interface (see for example the sensor simulator). Also, [[Distributed

Execution]], these functions identify which federate is producing the message. The functions are listed below. The arguments for all of these are identical to `printf` with the exception that a trailing newline is automatically added and therefore need not be included in the format string.

- `DEBUG_PRINT(format, ...)`: Use this for verbose messages that are only needed during debugging. Nothing is printed unless the target parameter `logging` is set to `debug`. This is a macro so that overhead is minimized when nothing is to be printed.
- `LOG_PRINT(format, ...)`: Use this for messages that are useful logs of the execution. Nothing is printed unless the target parameter `logging` is set to `log` or `debug`. This is a macro so that overhead is minimized when nothing is to be printed.
- `info_print(format, ...)`: Use this for messages that should normally be printed but may need to be redirected to a user interface such as a window or terminal (see `register_print_function` below). These messages can be suppressed by setting the logging target property to `warn` or `error`.
- `warning_print(format, ...)`: Use this for warning messages. These messages can be suppressed by setting the logging target property to `error`.
- `error_print(format, ...)`: Use this for error messages. These messages are not suppressed by any logging target property.
- `error_print_and_exit(format, ...)`: Use this for catastrophic errors.

In addition, a utility function is provided to register a function to redirect printed outputs:

- `register_print_function(function)`: Register a function that will be used instead of `printf` to print messages generated by any of the above functions. The function should accept the same arguments as `printf`.

Implementation Details

Included Libraries

The generated code includes the following standard C libraries, so there is no need for a reactor definition to explicitly include them if they are needed:

- stdio.h
- stdlib.h
- string.h

- time.h
- errno.h

In addition, the multithreaded implementation uses

- pthread.h

Single Threaded Implementation

The runtime library for the single-threaded implementation is in the following files:

- reactor.c
- reactor_common.c (included in the above using #include)
- pqueue.c

Three header files provide the interfaces:

- reactor.h
- ctarget.h
- pqueue.h

The strategy is to have two queues of pending accessor invocations, one that is sorted by timestamp (the event queue) and one that is sorted by priority (the reaction queue). Execution proceeds as follows:

1. At initialization, an event for each timer is put on the event queue and logical time is initialized to the current time, represented as the number of nanoseconds elapsed since January 1, 1970.

2. At each logical time, pull all events from event queue that have the same earliest time stamp, find the reactions that these events trigger, and put them on the reaction queue. If there are no events on the event queue, then exit the program (unless the `--keepalive true` command-line argument is given).

3. Wait until physical time matches or exceeds that earliest timestamp (unless the `--fast true` command-line argument is given). Then advance logical time to match that earliest timestamp.

4. Execute reactions in order of priority from the reaction queue. These reactions may produce outputs, which results in more events getting put on the reaction queue. Those reactions are assured of having lower priority than the reaction that is executing. If a reaction calls `schedule()`, an event will be put on the event queue, not the reaction queue.

5. When the reaction queue is empty, go to 2.

Multithreaded Implementation

The runtime library for the multithreaded implementation is in the following files:

- reactor_threaded.c
- reactor_common.c (included in the above using #include)
- pqueue.c

The same two header files provide the interfaces:

- reactor.h
- pqueue.h

The default number of worker threads is given by the `threads` argument in the target statement. This can be overridden with the `--threads` command-line argument.

Upon initialization, the main thread will create the specified number of worker threads. A good choice is for this number to match the number of available cores. Execution proceeds in a manner similar to the single threaded implementation except that the worker threads concurrently draw reactions from the reaction queue. The execution algorithm ensures that no reaction executes until all reactions that it depends on that are also on the reaction queue have executed at the current logical time.

FIXME: Describe the algorithm exploiting parallelism.

Reactors on Patmos

Reactors can be executed on Patmos, a bare-metal execution platform that is optimized for time-predictable execution. Well written C programs can be analyzed for their worst-case execution time (WCET).

Compiling and Running Reactors

Patmos can run in an FPGA, but there are also two simulators available:

1. `pasim` a software ISA simulator that is written in C++.
2. `patemu` a cycle-accurate hardware emulator generated from the hardware description.

To execute reactions on Patmos, the Patmos toolchain needs to be installed. The web page contains a quick start, detailed information including how to perform WCET analysis is available in the

Patmos Reference Handbook.

To execute the "hello world" reactor on Patmos use the LF compiler to generate the C code. Compile the reactor with the Patmos compiler (in `src-gen`):

```
patmos-clang Minimal.c -o Minimal.elf
```

The reactor can be executed on the SW simulator with:

```
pasim Minimal.elf
```

As Patmos is a bare metal runtime that has no notion of calendar time, its start time is considered the epoch and the following output will be observed:

```
Start execution at time Thu Jan  1 00:00:00 1970
plus 640000 nanoseconds.
Hello World.
Elapsed logical time (in nsec): 0
Elapsed physical time (in nsec): 3970000
```

The reactor can also be executed on the hardware emulator of Patmos:

```
patemu Minimal.elf
```

This execution is considerably slower than the SW simulator, as the concrete hardware of Patmos is simulated cycle-accurate.

Worst-Case Execution Time Analysis

Following example is a code fragment from Wcet.lf.

```
reactor Work {
    input in1: int;
    input in2: int;
    output out:int;
    reaction(in1, in2) -> out {=
        int ret;
        if (in1 > 10) {
                ret = in2 * in1;
        } else {
                ret = in2 + in1;
        }
        SET(out, ret);
    =}
}
```

We want to perform WCET analysis of the single reaction of the Work reactor. This reaction, depending on the input data, will either perform a multiplication, which is more expensive in Patmos, or an addition. The WCET analysis shall consider the multiplication path as the worst-case path. To generate the information for WCET analysis by the compiler we have to compile the application as follows:

```
patmos-clang -O2 -mserialize=wcet.pml Wcet.c
```

We investigate the C source code `Wcet.c` and find that the reaction we are interested is named `reaction_function1`. Therefore, we invoke WCET analysis as follows:

```
platin wcet -i wcet.pml -b a.out -e reaction_function1 --report
```

This results in following report:

```
...
[platin] INFO: Finished run WCET analysis (platin)          in 62 ms
[platin] INFO: best WCET bound: 242 cycles
---
- analysis-entry: reaction_function1
  source: platin
  cycles: 242
...
```

The analysis gives the WCET of 242 clock cycles for the reaction, which includes clock cycles for data cache misses. Further details on the WCET analysis tool `platin` and e.g., how to annotate loop bounds can be found in the Patmos Reference Handbook.

Note, that the WCET analysis of a reaction does only include the code of the reaction function, not the cache miss cost of calling the function from the scheduler or the cache miss cost when returning to the scheduler.

The CCpp Target

In some cases, it might be needed or desirable to mix C and C++ code, including in the body of reactions, or in included libraries and header files. The C target uses a C compiler by default, and will fail to compile mixed C/C++ language programs. As a remedy, we offer a `CCpp` target that uses the C runtime but employs a C++ compiler to compile your program. To use it, simply replace `target C` with `target CCpp`.

Here is a minimal example of a program written in the `CCpp` target, taken from HelloWorldCCPP.lf:

```
target CCpp;
reactor HelloWorld {
    preamble {=
        #include <iostream> // Note that no C++ header will be included by
    =}
    reaction(startup) {=
        std::cout << "Hello World." << std::endl;
    =}
}
main reactor {
    a = new HelloWorld();
}
```

Note: Unless some feature in the C target is needed, we recommend using the Cpp target that uses a runtime that is written natively in C++.

Note: A `.lf` file that uses the `CCpp` target cannot and should not be imported to an `.lf` file that uses the `C` target. Although these two targets use essentially the same runtime, such a scenario can cause unintended compiler errors.

Writing Reactors in C++

In the C++ reactor target for Lingua Franca, reactions are written in C++ and the code generator generates a standalone C++ program that can be compiled and run on all major platforms. Our continous integration ensures compatibility with Windows, MacOS and Linux. The C++ target solely depends on a working C++ build system including a recent C++ compiler (supporting C++17) and CMake (>= 3.5). It relies on the reactor-cpp runtime, which is automatically fetched and compiled in the background by the Lingua Franca compiler.

Note that C++ is not a safe language. There are many ways that a programmer can circumvent the semantics of Lingua Franca and introduce nondeterminism and illegal memory accesses. For example, it is easy for a programmer to mistakenly send a message that is a pointer to data on the stack. The destination reactors will very likely read invalid data. It is also easy to create memory leaks, where memory is allocated and never freed. Note, however, that the C++ reactor library is designed to prevent common errors and to encourage a safe modern C++ style. Here, we introduce the specifics of writing Reactor programs in C++ and present some guidelines for a style that will be safe.

Setup

The following tools are required in order to compile the generated C++ source code:

- A recent C++ compiler supporting C++17
- A recent version of cmake (At least 3.5)

A Minimal Example

A "hello world" reactor for the C++ target looks like this:

```
target Cpp;

main reactor {
    reaction(startup) {=
        std::cout << "Hello World!\n";
    =}
}
```

The `startup` action is a special action that triggers at the start of the program execution causing the reaction to execute. This program can be found in a file called Minimal.lf in the test directory,

where you can also find quite a few more interesting examples. If you compile this using the `lfc` command-line compiler or the Eclipse-based IDE, then generated source files will be put into a subdirectory called `src-gen/Minimal`. In addition, an executable binary will be compiled using your system's C++ compiler. The resulting executable will be called `Minimial` and be put in a subdirectory called `bin`. If you are in the C++ test directory, you can execute it in a shell as follows:

```
bin/Minimal
```

The resulting output should look something like this:

```
[INFO]  Starting the execution
Hello World!
[INFO]  Terminating the execution
```

The C++ Target Specification

To have Lingua Franca generate C++ code, start your `.lf` file with the following target specification:

```
target Cpp;
```

A C++ target specification may optionally include the following parameters:

- `build-type "type"`: The *build type* to be used by the C++ compiler. This can be any of Release, Debug, RelWithDebInfo and MinSizeRel. It defaults to Release.
- `cmake-include "file"`: An optional *file* to be included by the generated cmake build system. This gives control over the way LF programs are build and allows for instance to include and link to external libraries. (See `AsyncCallvack.lf` for an example)
- `compiler "command"`: The *command* to use to compile the generated code. Normally CMake selects the best compiler for your system, but you can use this parameter to point it to your preferred compiler.
- `external-runtime-path "path"`: When specified, the resulting binary links to a pre-compiled external runtime library located in *path* instead of the default one.
- `export-dependency-graph [true|false]`: Whether the compiled binary will export its internal dependency graph as a dot graph when executed. This is a debugging utility.
- `fast [true|false]`: Whether to execute as fast as possible ignoring real time. This defaults to false.
- `keepalive [true|false]`: Whether to continue executing even when there are no events on the event queue. The default is false. Usually, you will want to set this to true when you have

physical actions.

- `logging [ERROR|WARN|INF|DEBUG]`: The level of diagnostic messages about execution to print to the console. A message will print if this parameter is greater than or equal to the level of the message (`ERROR` < `WARN` < `INFO` < `DEBUG`).
- `no-compile [true|false]`: If this is set to true, then the Lingua Franca compiler will only generate code and not run the C++ compiler. This defaults to false.
- `no-runtime-validation [true|false]`: If this is set to true, then all runtime checks in reactor-cpp will be disabled. This brings a slight performance boost but should be used with care and only on tested programs. This defaults to false.
- `runtime-version "version"`: Specify the *version* of the runtime library the compiled binary should link against. *version* can be any tag, branch name or git hash in the reactor-cpp repository.
- `threads <n>`: The number of worker threads *n* that are used to execute reactions. This is required to be a positive integer. If this is not specified or set to zero, then the number of worker threads will be set to the number of hardware threads of the executing system. If this is set to one, the scheduler will not create any worker threads and instead inline the execution of reactions. This is an optimization and avoids any unnecessary synchronization. Note that, in contrast to the C target, the single threaded implementation is still thread safe and asynchronous reaction scheduling is supported.
- `timeout <n> <unit>`: This specifies to stop after the specified amount of logical time has elapsed. All events at that logical time that have microstep 0 will be processed, but not events with later tags. If any reactor calls `request_stop` before this logical time, then the program may stop at an earlier logical time. If no timeout is specified and `request_stop` is not called, then the program will continue to run until stopped by some other mechanism (such as Control-C).

Command-Line Arguments

The generated C++ program understands the following command-line arguments, each of which has a short form (one character) and a long form:

- `-f, --fast`: If set, then the program will execute as fast as possible, letting logical time advance faster than physical time.
- `-o, --timeout '<duration> <units>'`: Stop execution when logical time has advanced by the specified *duration*. The units can be any of nsec, usec, msec, sec, minute, hour, day, week, or the plurals of those.

- `-k, --keepalive`: If set, then the program will keep executing until either the `timeout` logical time is reached or the program is externally killed. If you have `physical action`s, it usually makes sense to set this.
- `-t, --threads <n>`: Use n worker threads for executing reactions.
- `-h, --help`: Print the above information.

If the main reactor declares parameters, these parameters will appear as additional CLI options that can be specified when invoking the binary (see Using Parameters).

Imports

The import statement can be used to share reactor definitions across several applications. Suppose for example that we modify the above Minimal.lf program as follows and store this in a file called HelloWorld.lf:

```
target Cpp;
reactor HelloWorld {
    reaction(startup) {=
        std::cout << "Hello World.\n";
    =}
}
main reactor HelloWorldTest {
    a = new HelloWorld();
}
```

This can be compiled and run, and its behavior will be identical to the version above. But now, this can be imported into another reactor definition as follows:

```
target Cpp;
import HelloWorld.lf;
main reactor TwoHelloWorlds {
    a = new HelloWorld();
    b = new HelloWorld();
}
```

This will create two instances of the HelloWorld reactor, and when executed, will print "Hello World" twice.

Note that in the above example, the order in which the two reactions are invoked is undefined because there is no causal relationship between them. In fact, you might see garbled output as on default multiple worker threads are used to execute the program and `std::cout` is not thread safe. You can restrict execution to one thread if you modify the target specification to say:

```
target Cpp {threads: 1};
```

A more interesting illustration of imports can be found in the Import.lf test case.

Preamble

Reactions may contain arbitrary C++ code, but often it is convenient for that code to invoke external libraries or to share type and/or method definitions. For either purpose, a reactor may include a **preamble** section. For example, the following reactor uses `atoi` from the common `stdlib` C library to convert a string to an integer:

```
main reactor Preamble {
    private preamble {=
        include <cstdlib>
    =}
    timer t;
    reaction(t) {=
        char* s = "42";
        int i = atoi(s);
        std::cout << "Converted string << s << " to nt " << i << '\n';
    =}
}
```

This will print:

```
Converted string 42 to int 42.
```

By putting the `#include` in the **preamble**, the library becomes available in all reactions of this reactor. Note the **private** qualifier before the **preamble** keyword. This ensures that the preamble is only visible to the reactions defined in this reactor and not to any other reactors. In contrast, the **public** qualifier ensures that the preamble is also visible to other reactors in files that import the reactor defining the public preamble.

See for instance the reactor in Preamble.lf:

```
reactor Preamble {
    public preamble {=
        struct MyStruct {
            int foo;
            std::string bar;
        };
    =}

    private preamble {=
        int add_42(int i) {
            return i + 42;
        }
    =}

    logical action a:MyStruct;

    reaction(startup) {=
        a.schedule({add_42(42), "baz"});
    =}

    reaction(a) {=
        auto& value = *a.get();
        std::cout << "Received " << value.foo << " and '" << value.bar << "
    =}
}
```

It defines both, a public and a private preamble. The public preamble defines the type `MyStruct`. This type definition will be visible to all elements of the `Preamble` reactor as well as to all reactors defined in files that import `Preamble`. The private preamble defines the function `add_42(int i)`. This function will only be usable to reactions within the `Preamble` reactor.

You can think of public and private preambles as the equivalent of header files and source files in C++. In fact, the public preamble will be translated to a header file and the private preamble to a source file. As a rule of thumb, all types that are used in port or action definitions as well as in state variables or parameters should be defined in a public preamble. Also declarations of functions to be shared across reactors should be placed in the public preamble. Everything else, like function definitions or types that are used only within reactions should be placed in a private preamble.

Note that preambles can also be specified on the file level. These file level preambles are visible to all reactors within the file. An example of this can be found in PreambleFile.lf.

Admittedly, the precise interactions of preambles and imports can become confusing. The preamble mechanism will likely be refined in future revisions.

Note that functions defined in the preamble cannot access members such as state variables of the reactor unless they are explicitly passed as arguments. If access to the inner state of a reactor is required, methods present a viable and easy to use alternative.

Reactions

Recall that a reaction is defined within a reactor using the following syntax:

```
reaction(triggers) uses -> effects {=
  ... target language code ...
=}
```

In this section, we explain how **triggers**, **uses**, and **effects** variables work in the C++ target.

Inputs and Outputs

In the body of a reaction in the C++ target, the value of an input is obtained using the syntax `*name.get()`, where `name` is the name of the input port. Note that `get()` always returns a pointer to the actual value. Thus the pointer needs to be dereferenced with `*` to obtain the value. (See Sending and Receiving Large Data Types for an explanation of the exact mechanisms behind this pointer access). To determine whether an input is present, `name.is_present()` can be used. Since `get()` returns a `nullptr` if no value is present, `name.get() != nullptr` can be used alternatively for checking presence.

For example, the Determinism.lf test case in the test directory includes the following reactor:

```
reactor Destination {
    input x:int;
    input y:int;
    reaction(x, y) {=
        int sum = 0;
        if (x.is_present()) {
            sum += *x.get();
        }
        if (y.is_present()) {
            sum += *y.get();
        }
        std::cout << "Received " << sum << std::endl;
    =}
}
```

The reaction refers to the inputs `x` and `y` and tests for the presence of values using `x.is_present()` and `y.is_present()`. If a reaction is triggered by just one input, then normally it is not necessary to test for its presence; it will always be present. But in the above example, there are two triggers, so the reaction has no assurance that both will be present.

Inputs declared in the **uses** part of the reaction do not trigger the reaction. Consider this modification of the above reaction:

```
reaction(x) y {=
    int sum = *x.get();
    if (y.is_present()) {
        sum += *y.get();
    }
    std::cout << "Received " << sum << std::endl;
=}
```

It is no longer necessary to test for the presence of `x` because that is the only trigger. The input `y`, however, may or may not be present at the logical time that this reaction is triggered. Hence, the code must test for its presence.

The **effects** portion of the reaction specification can include outputs and actions. Actions will be described below. Outputs are set using a `set()` method on an output port. For example, we can further modify the above example as follows:

```
output z:int;
reaction(x) y -> z {=
    int sum = *x.get();
    if (y.is_present()) {
        sum += *y.get();
    }
    z.set(sum);
=}
```

If an output gets set more than once at any logical time, downstream reactors will see only the *final* value that is set. Since the order in which reactions of a reactor are invoked at a logical time is deterministic, and whether inputs are present depends only on their timestamps, the final value set for an output will also be deterministic.

An output may even be set in different reactions of the same reactor at the same logical time. In this case, one reaction may wish to test whether the previously invoked reaction has set the output. It can check `name.is_present()` to determine whether the output has been set. For example, the following reactor (see TestForPreviousOutput.lf) will always produce the output 42:

```
reactor Source {
    output out:int;
    reaction(startup) -> out {=
        // Set a seed for random number generation based on the current tim
        std::srand(std::time(nullptr));
        // Randomly produce an output or not.
        if (std::rand() % 2) {
            out.set(21);
        }
    =}
    reaction(startup) -> out {=
        if (out.is_present()) {
            int previous_output = *out.get();
            out.set(2 * previous_output);
        } else {
            out.set(42);
        }
    =}
}
```

The first reaction may or may not set the output to 21. The second reaction doubles the output if it has been previously produced and otherwise produces 42.

Using State Variables

A reactor may declare state variables, which become properties of each instance of the reactor. For example, the following reactor (see Count.lf and CountTest.lf) will produce the output sequence 1, 2, 3, ... :

```
reactor Count {
    state i:int(0);
    output c:int;
    timer t(0, 1 sec);
    reaction(t) -> c {=
        i++;
        c.set(i);
    =}
}
```

The declaration on the second line gives the variable the name `count`, declares its type to be `int`, and initializes its value to 0. The type and initial value can be enclosed in the C++-code delimiters `{= ... =}` if they are not simple identifiers, but in this case, that is not necessary.

In the body of the reaction, the state variable is automatically in scope and can be referenced directly by its name. Since all reactions, state variables and also parameters of a reactor are members of the same class, reactions can also reference state variables (or parameters) using the this pointer: `this->name`.

A state variable may be a time value, declared as follows:

```
state time_value:time(100 msec);
```

The type of the generated `time_value` variable will be `reactor::Duration`, which is an alias for `std::chrono::nanoseconds`.

For the C++ target, Lingua Franca provides two alternative styles for initializing state variables. We can write `state foo:int(42)` or `state foo:int{42}`. This allows to distinguish between the different initialization styles in C++. `foo:int(42)` will be translated to `int foo(42)` and `foo:int{42}` will be translated to `int foo{42}` in the generated code. Generally speaking, the `{...}` style should be preffered in C++, but it is not always applicable. Hence we allow the LF programmer to choose the style. Due to the peculiarities of C++, this is particularly important for more complex data types. For instance, `state foo:std::vector<int>(4,2)` would be initialized to the list `[2,2,2,2]` whereas `state foo:std::vector<int>{4,2}` would be initialized to the list `[4,2]`.

State variables can have array values. For example, the [MovingAverage] (https://github.com/lf-lang/lingua-franca/blob/master/test/Cpp/src/MovingAverage.lf) reactor computes the **moving average** of the last four inputs each time it receives an input:

```
reactor MovingAverageImpl {
    state delay_line:double[3]{0.0, 0.0, 0.0};
    state index:int(0);
    input in:double;
    output out:double;

    reaction(in) -> out {=
        // Calculate the output.
        double sum = *in.get();
        for (int i = 0; i < 3; i++) {
            sum += delay_line[i];
        }
        out.set(sum/4.0);

        // Insert the input in the delay line.
        delay_line[index] = *in.get();

        // Update the index for the next input.
        index++;
        if (index >= 3) {
            index = 0;
        }
    =}
}
```

The second line declares that the type of the state variable is an fixed-size array of 3 `double`s with the initial value of the being filled with zeros (note the curly braces). If the size is given in the type specification, then the code generator will declare the type of the state variable using `std::array`. In the example above, the type of `delay_line` is `std::array<3, double>`. If the size specifier is omitted (e.g. `state x:double[]`). The code generator will produce a variable-sized array using `std::vector`.

State variables with more complex types such as classes or structs can be similiarly initialized. See StructAsState.lf.

Using Parameters

Reactor parameters work similar to state variables in C++. However, they are always declared as `const` and initialized during reactor instantiation. Thus, the value of a parameter may not be changed. For example, the Stride reactor modifies the above `Count` reactor so that its stride is a parameter:

```
reactor Count(stride:int(1)) {
    state count:int(0);
    output y:int;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        y.set(count);
        count += stride;
    =}
}
reactor Display {
    input x:int;
    reaction(x) {=
        std::cout << "Received " << *x.get() << std::endl;
    =}
}
main reactor Stride {
    c = new Count(stride = 2);
    d = new Display();
    c.y -> d.x;
}
```

The first line defines the `stride` parameter, gives its type, and gives its initial value. As with state variables, the type and initial value can be enclosed in `{= ... =}` if necessary.

When the reactor is instantiated, the default parameter value can be overridden. This is done in the above example near the bottom with the line:

```
c = new Count(stride = 2);
```

If there is more than one parameter, use a comma separated list of assignments.

Also parameters can have fixed- or variable-sized array values. The ArrayAsParameter example outputs the elements of an array as a sequence of individual messages:

```
reactor Source(sequence:int[]{0, 1, 2}) {
    output out:int;
    state count:int(0);
    logical action next:void;
    reaction(startup, next) -> out, next {=
        out.set(sequence[count]);
        count++;
        if (count < sequence.size()) {
            next.schedule();
        }
    =}
}
```

The **logical action** named `next` and the `schedule` method are explained below in Scheduling Delayed Reactions; here they are used simply to repeat the reaction until all elements of the array have been sent. Note that similiar aas for state variables, curly braces `{...}` can optionally be used for initialization.

Note that also the main reactor can be parameterized:

```
main reactor Hello(msg: std::string("World")) {
    reaction(startup) {=
        std::cout << "Hello " << msg << "!\n";
    =}
}
```

This program will print "Hello World!" by default. However, since `msg` is a main reactor parameter, the C++ code generator will extend the CLI argument parser and allow to overwrite `msg` when invoking the program. For instance,

```
bin/Hello --msg Earth
```

will result in "Hello Earth!" being printed.

Using Methods

Sometimes reactors need to perform certain operations on state variables and/or parameters that are shared between reactions or that are too complex to be implemented in a single reaction. In such cases, methods can be defined within reactors to facilitate code reuse and enable a better structuring of the reactor's functionality. Analogous to class methods, methods in LF can access all state variables and parameters, and can be invoked from all reaction bodies or from other methods. Consdider the Method example:

```
main reactor {
    state foo:int(2);

    const method getFoo(): int {=
        return foo;
    =}

    method add(x:int) {=
        foo += x;
    =}

    reaction(startup){=
        std::cout << "Foo is initialized to " << getFoo() << '\n';
        add(40);
        std::cout << "2 + 40 = " << getFoo() << '\n';
    =}
}
```

This reactor defines two methods `getFoo` and `add`. `getFoo` is quailfied as a const method, which indicates that it has read-only access to the state variables. This is direclty translated to a C++ const method in the code generation process. `getFoo` receives no arguments and returns an integer (`int`) indicating the current value of the `foo` state variable. `add` returns nothing (`void`) and receives one interger argument, which it uses to increment `foo`. Both methods are visible in all reactions of the reactor. In this example, the reactio to startup calles both methods in order ro read and modify its state.

Sending and Receiving Large Data Types

You can define your own datatypes in C++ or use types defined in a library and send and receive those. Consider the StructAsType example:

```
reactor StructAsType {
    public preamble {=
        struct Hello {
            std::string name;
            int value;
        };
    =}

    output out:Hello;
    reaction(startup) -> out {=
        Hello hello{"Earth, 42};
        out.set(hello);
    =}
}
```

The **preamble** code defines a struct datatype. In the reaction to **startup**, the reactor creates an instance of this struct on the stack (as a local variable named `hello`) and then copies that instance to the output using the `set()` method. For this reason, the C++ reactor runtime provides more sophisticated ways to allocate objects and send them via ports.

The C++ library defines two types of smart pointers that the runtime uses internally to implement the exchange of data between ports. These are `reactor::MutableValuePtr<T>` and `reactor::ImmutableValuePtr<T>`. `reactor::MutableValuePtr<T>` is a wrapper around `std::unique_ptr` and provides read and write access to the value hold, while ensuring that the value has a unique owner. In contrast, `reactor::ImmutableValuePtr<T>` is a wrapper around `std::shared_pointer` and provides read only (const) access to the value it holds. This allows data to be shared between reactions of various reactors, while guarantee data consistency. Similar to `std::make_unique` and `std::make_shared`, the reactor library provides convenient function for creating mutable and immutable values pointers: `reactor::make_mutable_value<T>(...)` and `reactor::make_immutable_value<T>(...)`.

In fact this code from the example above:

```
Hello hello{"Earth, 42};
out.set(hello);
```

implicitly invokes `reactor::make_immutable_value<Hello>(hello)` and could be rewritten as

```
Hello hello{"Earth, 42};
out.set(reactor::make_immutable_value<Hello>(hello));
```

This will invoke the copy constructor of `Hello`, copying its content from the `hello` instance to the newly created `reactor::ImmutableValuePtr<Hello>`.

Since copying large objects is inefficient, the move semantics of C++ can be used to move the ownership of object instead of copying it. This can be done in the following two ways. First, by directly creating a mutable or immutable value pointer, where a mutable pointer allows modification of the object after it has been created:

```
auto hello = reactor::make_mutable_value<Hello>("Earth", 42);
hello->name = "Mars";
out.set(std::move(hello));
```

An example of this can be found in StructPrint.lf. Not that after the call to `std::move`, hello is `nullptr` and the reaction cannot modify the object anymore. Alternatively, if no modification is requires, the object can be instantiated directly in the call to `set()` as follows:

```
out.set({"Earth", 42});
```

An example of this can be found in StructAsTypeDirect.

Getting a value from an input port of type `T` via `get()` always returns an `reactor::ImmutableValuePtr<T>`. This ensures that the value cannot be modified by multiple reactors receiving the same value, as this could lead to an inconsistent state and nondeterminism in a multi-threaded execution. An immutable value pointer can be converted to a mutable pointer by calling `get_mutable_copy`. For instance, the ArrayScale reactor modifies elements of the array it receives before sending it to the next reactor:

```
reactor Scale(scale:int(2)) {
    input in:int[3];
    output out:int[3];

    reaction(in) -> out {=
        auto array = in.get().get_mutable_copy();
        for(int i = 0; i < array->size(); i++) {
            (*array)[i] = (*array)[i] * scale;
        }
        out.set(std::move(array));
    =}
}
```

Currently `get_mutable_copy()` always copies the contained value to safely create a mutable pointer. However, a future implementation could optimize this by checking if any other reaction is accessing the same value. If not, the value can simply be moved from the immutable pointer to a mutable one.

Timed Behavior

Timers are specified exactly as in the Lingua Franca language specification. When working with time in the C++ code body of a reaction, however, you will need to know a bit about its internal representation.

The Reactor C++ library uses `std::chrono` for representing time. Specifically, the library defines two types for representing durations and timepoints: `reactor::Duration` and `reactor::TimePoint`. `reactor::Duration` is an alias for `std::chrono::nanosecods`. `reactor::TimePoint` is alias for `std::chrono::time_point<std::chrono::system_clock, std::chrono::nanoseconds>`. As you can see from these definitions, the smallest time step that can be represented is one nanosecond. Note that `reactor::TimePoint` describes a specific point in time and is associated with a specific clock, whereas `reactor::Duration` defines a time interval between two time points.

Lingua Franca uses a superdense model of logical time. A reaction is invoked at a logical **tag**. In the C++ library, a tag is represented by the class `reactor::Tag`. In essence, this class is a tuple of a `reactor::TimePoint` representing a specific point in logical time and a microstep value (of type `reactor::mstep_t`, which is an alias for `unsigned long`). `reactor::Tag` provides two methods for getting the time point or the microstep:

```
const TimePoint& time_point() const;
const mstep_t& micro_step() const;
```

The C++ code in reaction bodies has access to library functions that allow to retrieve the current logical or physical time:

- `TimePoint get_physical_time()`: Get the current physical time.
- `TimePoint get_logcial_time()`: Get the current logical time.
- `Duration get_elapsed_physical_time()`: Get the physical time elapsed since program start.
- `Duration get_elapsed_logical_time()`: Get the logical time elapsed since program start.

A reaction can examine the current logical time (which is constant during the execution of the reaction). For example, consider the GetTime example:

```
main reactor {
    timer t(0, 1 sec);
    reaction(t) {=
        auto logical = get_logical_time();
        std::cout << "Logical time is " << logical << std::endl;
    =}
}
```

Note that the `<<` operator is overloaded for both `reactor::TimePoint` and `reactor::Duration` and will print the time information accordingly.

When executing the above program, you will see something like this:

```
[INFO]  Starting the execution
Logical time is 2021-05-19 14:06:09.496828396
Logical time is 2021-05-19 14:06:10.496828396
Logical time is 2021-05-19 14:06:11.496828396
Logical time is 2021-05-19 14:06:11.496828396
...
```

If you look closely, you will see that each printed logical time is one second larger than the previous one.

You can also obtain the *elapsed* logical time since the start of execution:

```
main reactor {
    timer t(0, 1 sec);
    reaction(t) {=
        auto elapsed = get_elapsed_logical_time();
        std::cout << "Elapsed logical time is " << elapsed << std::endl;
        std::cout << "In seconds: " <<  std::chrono::duration_cast<std::chr
    =}
}
```

Using `std::chrono` it is also possible to convert between time units and directly print the number of elapsed seconds as seen above. The resulting output of this program will be:

```
[INFO]  Starting the execution
Elapsed logical time is 0 nsecs
In seconds: 0 secs
Elapsed logical time is 1000000000 nsecs
In seconds: 1 secs
Elapsed logical time is 2000000000 nsecs
In seconds: 2 secs
...
```

You can also get physical and elapsed physical time:

```
main reactor {
    timer t(0, 1 sec);
        reaction(t) {=
        auto logical = get_logical_time();
        auto physical = get_physical_time();
                auto elapsed = get_elapsed_physical_time();
        std::cout << "Physical time is " << physical << std::endl;
        std::cout << "Elapsed physical time is " << elapsed << std::endl;
        std::cout << "Time lag is " << physical - logical << std::endl;
   =}
}
```

Notice that the physical times are increasing by *roughly* one second in each reaction. The output also shows the lag between physical and logical time. If you set the `fast` target parameter to `true`, then physical time will elapse much faster than logical time. The above program will produce something like this:

```
[INFO]  Starting the execution
Physical time is 2021-05-19 14:25:18.070523014
Elapsed physical time is 2601601 nsecs
Time lag is 2598229 nsecs
Physical time is 2021-05-19 14:25:19.068038275
Elapsed physical time is 1000113888 nsecs
Time lag is 113490 nsecs
[INFO]  Physical time is Terminating the execution
2021-05-19 14:25:20.068153026
Elapsed physical time is 2000228689 nsecs
Time lag is 228241 nsecs
```

For specifying time durations in code chrono provides convenient literal operators in `std::chrono_literals`. This namespace is automatically included for all reaction bodies. Thus, we can simply write:

```
std::cout << 42us << std::endl;
std::cout << 1ms << std::endl;
std::cout << 3s << std::endl;
```

which prints:

```
42 usecs
1 msecs
3 secs
```

Scheduling Delayed Reactions

The C++ provides a simple interface for scheduling actions via a `schedule()` method. Actions are described in the Language Specification document. Consider the Schedule reactor:

```
reactor Schedule {
        input x:int;
    logical action a;
    reaction(a) {=
         auto elapsed_time = get_elapsed_logical_time();
         std::cout << "Action triggered at logical time " << elapsed_time.c
                 << " after start" << std::endl; elapsed_time);
    =}
    reaction(x) -> a {=
        a.schedule(200ms);
    =}
}
```

When this reactor receives an input `x`, it calls `schedule()` on the action `a`, specifying a logical time offset of 200 milliseconds. The action `a` will be triggered at a logical time 200 milliseconds after the arrival of input `x`. At that logical time, the second reaction will trigger and will use the `get_elapsed_logical_time()` function to determine how much logical time has elapsed since the start of execution.

Notice that after the logical time offset of 200 msec, there may be another input `x` simultaneous with the action `a`. Because the reaction to `a` is given first, it will execute first. This becomes important when such a reactor is put into a feedback loop (see below).

TODO: Explain physical actions as well!

Zero-Delay Actions

If the specified delay in a `schedule()` is omitted or is zero, then the action `a` will be triggered one **microstep** later in **superdense time** (see Superdense Time). Hence, if the input `x` arrives at metric logical time *t*, and you call `schedule()` in one of the following ways:

```
a.schedule();
a.schedule(0s);
a.schedule(reactor::Duration::zero());
```

then when the reaction to `a` is triggered, the input `x` will be absent (it was present at the *previous* microstep). The reaction to `x` and the reaction to `a` occur at the same metric time *t*, but separated by one microstep, so these two reactions are *not* logically simultaneous.

As discussed above the he metric time is visible to the rogrammer and can be obtained in a reaction using either `get_elapsed_logical_time()` or `get_logical_time()`.

As described in the Language Specification document, action declarations can have a *min_delay* parameter. This modifies the timestamp further. Also, the action declaration may be **physical** rather than **logical**, in which case, the assigned timestamp will depend on the physical clock of the executing platform.

Actions With Values

If an action is declared with a data type, then it can carry a **value**, a data value that becomes available to any reaction triggered by the action. This is particularly useful for physical actions that are externally triggered because it enables the action to convey information to the reactor. This could be, for example, the body of an incoming network message or a numerical reading from a sensor.

Recall from the Contained Reactors section in the Language Specification document that the **after** keyword on a connection between ports introduces a logical delay. This is actually implemented using a logical action. We illustrate how this is done using the DelayInt example:

```
reactor Delay(delay:time(100 msec)) {
    input in:int;
    output out:int;
    logical action d:int;
    reaction(in) -> d {=
        d.schedule(in.get(), delay);
    =}
    reaction(d) -> out {=
        if (d.is_present()) {
            out.set(d.get());
        }
    =}
}
```

Using this reactor as follows

```
    d = new Delay();
    source.out -> d.in;
    d.in -> sink.out
```

is equivalent to

```
    source.out -> sink.in after 100 msec
```

(except that our `Delay` reactor will only work with data type `int`).

The action `d` is specified with a type `int`. The reaction to the input `in` declares as its effect the action `d`. This declaration makes it possible for the reaction to schedule a future triggering of `d`. In the C++ target, actions use the same mechanism for passing data via value pointers as do ports. In the example above, the `reactor::ImmutablValuePtr<int>` derived by the call to `in.get()` is passed directly to `schedule()`. Similarly, the value can later be retrieved from the action with `d.get()` and passed to the output port.

The first reaction declares that it is triggered by `d` and has effect `out`. Because this reaction is first, the `out` at any logical time can be produced before the input `in` is even known to be present. Hence, this reactor can be used in a feedback loop, where `out` triggers a downstream reactor to send a message back to `in` of this same reactor. If the reactions were given in the opposite order, there would be causality loop and compilation would fail.

If you are not sure whether an action carries a value, you can test for it using `is_present()`:

```
reaction(d) -> out {=
    if (d.is_present()) {
        out.set(d.get());
    }
=}
```

It is possible to both be triggered by and schedule an action the same reaction. For example, this reactor will produce a counting sequence after it is triggered the first time:

```
reactor CountSelf(delay:time(100 msec)) {
    output out:int;
    logical action a:int;
    reaction(startup) -> a, out {=
        out.set(0);
        a.schedule_int(1, delay);
    =}
    reaction(a) -> a, out {=
        out.set(a.get());
        a.schedule_int(*a.get() + 1, delay);
    =}
}
```

Of course, to produce a counting sequence, it would be more efficient to use a state variable.

Stopping Execution

A reaction may request that the execution stops after all events with the current timestamp have been processed by calling `environment()->sync_shutdown()`. There is also a method `environment()->async_shutdown()` which may be invoked from outside an reaction, like an external thread.

Log and Debug Information

The reactor-cpp library provides logging utilities in logging.hh for producing messages to be made visible when the generated program is run. Of course `std::cout` or `printf` can be used for the same purpose, but the logging mechanism provided by reactor-cpp is thread-safe ensuring that messages produced in parallel reactions are not interleaved with each other and provides common way for turning messages of a certain severity on and off.

In particular, reactor-cpp provides the following logging interfaces:

- `reactor::Debug()`: for verbose debug messages
- `reactor::Info()`: for info messages of general interest, info is the default severity level
- `reactor::Warning()`: for warning messages
- `reactor::Error()`: for errors

These utilities can be used analogues to `std::cout`. For instance:

```
reactor::Info() << "Hello World! It is " << get_physical_time();
```

Note that unlike `std::cout` the new line delimiter is automatically added to the end of the message.

Which type of messages are actually produced by the compiled program can be controlled with the `log-level` target property.

Writing Reactors in TypeScript

In the TypeScript reactor target for Lingua Franca, reactions are written in TypeScript and the code generator generates a standalone TypeScript program that can be compiled to JavaScript and run on Node.js.

TypeScript reactors bring the strengths of TypeScript and Node.js to Lingua Franca programming. The TypeScript language and its associated tools enable static type checking for both reaction code and Lingua Franca elements like ports and actions. The Node.js JavaScript runtime provides an execution environment for asynchronous network applications. With Node.js comes Node Package Manager (npm) and its large library of supporting modules.

In terms of raw performance on CPU intensive operations, TypeScript reactors are about two orders of magnitude slower than C reactors. But excelling at CPU intensive operations isn't really the point of Node.js (or by extension TypeScript reactors). Node.js is about achieving high throughput on network applications by efficiently handling asynchronous I/O operations. Keep this in mind when choosing the right Lingua Franca target for your application.

Setup

First, make sure Node.js is installed on your machine. You can download Node.js here. The npm package manager comes along with Node.

After installing Node, you may optionally install the TypeScript compiler.

```
npm install -g typescript
```

TypeScript reactor projects are created with a local copy of the TypeScript compiler, but having the TypeScript compiler globally installed can be useful for debugging type errors and type checking on the command line.

A Minimal Example

A "hello world" reactor for the TypeScript target looks like this:

```
target TypeScript;
main reactor Minimal {
    timer t;
    reaction(t) {=
        console.log("Hello World.");
    =}
}
```

The timer triggers at the start time of the execution causing the reaction to execute. This program can be found in a file called `Minimal.lf` in the test directory, where you can also find quite a few more interesting examples. If you compile this using the `lfc` command-line compiler or the Eclipse-based IDE, a number of files and directories will be generated. You can run the compiled JavaScript program (from `Minimal.lf`'s directory) with the command:

```
$ node Minimal/dist/Minimal.js
```

The resulting output should look something like this:

```
Hello World.
```

Notice the compiler generates a project directory with the name of the .lf file. In this example the .lf file's name is "Minimal" but more generally, for `<LF_file_name>.lf` the command to run the program is:

```
$ node <LF_file_name>/dist/<LF_file_name>.js
```

Refer to the TypeScript Project Structure section to learn why the command looks like this.

The TypeScript Target Specification

To have Lingua Franca generate TypeScript code, start your `.lf` file with the following target specification:

```
target TypeScript;
```

A TypeScript target specification may optionally include the following parameters:

- `fast [true|false]`: Whether to execute as fast as possible ignoring real time. This defaults to false.
- `keepalive [true|false]`: Whether to continue executing even when there are no events on the event queue. The default is false. Usually, you will want to set this to true when you have **physical action**s.

- `logging [ERROR|WARN|INFO|LOG|DEBUG]`: The level of diagnostic messages about execution to print to the console. A message will print if this parameter is greater than or equal to the level of the message (`ERROR` < `WARN` < `INFO` < `LOG` < `DEBUG`). Internally this is handled by the ulog module.
- `timeout <n> <units>`: The amount of logical time to run before exiting. By default, the program will run forever or until forcibly stopped, with control-C, for example.

For example, for the TypeScript target, in a source file named `Foo.lf`, you might specify:

```
target TypeScript {
    fast: true,
    timeout: 10 secs,
    logging: INFO,

};
```

The `fast` option given above specifies to execute the file as fast as possible, ignoring timing delays.

The `logging` option indicates diagnostic messages tagged as `ERROR`, `WARN`, and `INFO` should print to the console. Messages tagged `LOG` or `DEBUG` will not print.

The `timeout` option specifies to stop after 10 seconds of logical time have elapsed.

Command-Line Arguments

The generated JavaScript program understands the following command-line arguments, each of which has a short form (one character) and a long form:

- `-f, --fast [true | false]`: Specifies whether to wait for physical time to match logical time. The default is `false`. If this is `true`, then the program will execute as fast as possible, letting logical time advance faster than physical time.
- `-o, --timeout '<duration> <units>'`: Stop execution when logical time has advanced by the specified *duration*. The units can be any of nsec, usec, msec, sec, minute, hour, day, week, or the plurals of those. For the duration and units of a timeout argument to be parsed correctly as a single value, these should be specified in quotes with no leading or trailing space (eg '5 sec').
- `-k, --keepalive [true | false]`: Specifies whether to stop execution if there are no events to process. This defaults to `false`, meaning that the program will stop executing when there are no more events on the event queue. If you set this to `true`, then the program will

keep executing until either the `timeout` logical time is reached or the program is externally killed. If you have `physical action`s, it usually makes sense to set this to `true`.

- `-l, --logging [ERROR | WARN | INFO | LOG | DEBUG]`: The level of logging messages from the reactor-ts runtime to to print to the console. Messages tagged with a given type (error, warn, etc.) will print if this argument is greater than or equal to the level of the message (`ERROR` < `WARN` < `INFO` < `LOG` < `DEBUG`).
- `-h, --help`: Print this usage guide. The program will not execute if this flag is present.

If provided, a command line argument will override whatever value the corresponding target property had specified in the source .lf file.

Command line options are parsed by the command-line-arguments module with these rules. For example

```
$ node <LF_file_name>/dist/<LF_file_name>.js -f false --keepalive=true -o '
```

is a valid setting.

Any errors in command-line arguments result in printing the above information. The program will not execute if there is a parsing error for command-line arguments.

Custom Command-Line Arguments

User-defined command-line arguments may be created by giving the main reactor parameters. Assigning the main reactor a parameter of type `string`, `number`, `boolean`, or `time` will add an argument with corresponding name and type to the generated program's command-line-interface. Custom arguments will also appear in the generated program's usage guide (from the `--help` option). If the generated program is executed with a value specified for a custom command-line argument, that value will override the default value for the corresponding parameter. Arguments typed `string`, `number`, and `boolean` are parsed in the expected way, but `time` arguments must be specified on the command line like the `--timeout` property as `'<duration> <units>'` (in quotes).

Note: Custom arguments may not have the same names as standard arguments like `timeout` or `keepalive`.

For example this reactor has a custom command line argument named `customArg` of type `number` and default value `2`:

```
target TypeScript;
main reactor clArg(customArg:number(2)) {
    reaction (startup) {=
        console.log(customArg);
    =}
}
```

If this reactor is compiled from the file `simpleCLArgs.lf`, executing

```
node simpleCLArgs/dist/simpleCLArgs.js
```

outputs the default value `2`. But running

```
node simpleCLArgs/dist/simpleCLArgs.js --customArg=42
```

outputs `42`. Additionally, we can view documentation for the custom command line argument with the `--help` command.

```
node simpleCLArgs/dist/simpleCLArgs.js -h
```

The program will generate the standard usage guide, but also

```
--customArg '<duration> <units>'                    Custom argument. Refer
                                                      <path>/simpleCLArgs.l
                                                      for documentation.
```

Additional types for Custom Command-Line Arguments

Main reactor parameters that are not typed `string`, `number`, `boolean`, or `time` will not create custom command-line arguments. However, that doesn't mean it is impossible to obtain other types from the command line, just use a `string` and specify how the parsing is done yourself. See below for an example of a reactor that parses a custom command-line argument of type `string` into a state variable of type `Array<number>` using `JSON.parse` and a user-defined type guard.

```
target TypeScript;
main reactor customType(arrayArg:string("")) {
    preamble {=
        function isArrayOfNumbers(x: any): x is Array<number> {
            for (let item of x) {
                if (typeof item !== "number") {
                    return false;
                }
            }
            return true;
        }
    =}
    state foo:{=Array<number>=}({=[]=});
    reaction (startup) {=
        let parsedArgument = JSON.parse(customType);
        if (isArrayOfNumbers(parsedArgument)) {
            foo = parsedArgument;
            }
        else {
            throw new Error("Custom command line argument is not an array o
        }
        console.log(foo);
    =}
}
```

Imports

The import statement can be used to share reactor definitions across several applications. Suppose for example that we modify the above `Minimal.lf` program as follows and store this in a file called `HelloWorld.lf`:

```
target TypeScript;
reactor HelloWorldInside {
    timer t;
    reaction(t) {=
        console.log("Hello World.");
    =}
}
main reactor HelloWorld {
    a = new HelloWorldInside();
}
```

This can be compiled and run, and its behavior will be identical to the version above. But now, this can be imported into another reactor definition as follows:

```
target TypeScript;
import HelloWorld.lf;
main reactor TwoHelloWorlds {
    a = new HelloWorldInside();
    b = new HelloWorldInside();
}
```

This will create two instances of the HelloWorld reactor, and when executed, will print "Hello World" twice.

A more interesting illustration of imports can be found in the `Import.lf` test case in the test directory.

Preamble

Reactions may contain arbitrary TypeScript code, but often it is convenient for that code to invoke node modules or to share function/type/class definitions. For these purposes, a reactor may include a **preamble** section. For example, the following reactor uses Node's built-in path module to extract the base name from a path:

```
target TypeScript;
main reactor Preamble {
    preamble {=
        import * as path from 'path';
    =}
    reaction (startup) {=
        var filename = path.basename('/Users/Refsnes/demo_path.js');
        console.log(filename);
    =}
}
```

This will print:

```
demo_path.js
```

By putting the `import` in the **preamble**, the library becomes available in all reactions of this reactor. Oddly, it also becomes available in all subsequently defined reactors in the same file. It's a bit more complicated to set up node.js modules from npm that aren't built-in, but the reaction code to `import` them is the same as what you see here.

You can also use the preamble to define functions that are shared across reactions and reactors:

```
main reactor Preamble {
    preamble {=
        function add42( i:number) {
            return i + 42;
        }
    =}
    timer t;
    reaction(t) {=
        let s = "42";
        let radix = 10;
        let i = parseInt(s, radix);
        console.log("Converted string " + s + " to number " + i);
        console.log("42 plus 42 is " + add42(42));
    =}
}
```

Not surprisingly, this will print:

```
Converted string 42 to number 42
42 plus 42 is 84
```

Using Node Modules

Installing Node.js modules for TypeScript reactors with `npm` is essentially the same as installing modules for an ordinary Node.js program. First, write a Lingua Franca program (`Foo.lf`) and compile it. It may not type check if if you're importing modules in the preamble and you haven't installed the modules yet, but compiling your program will cause the TypeScript code generator to produce a project for your program. There should now be a package.json file in the same directory as your .lf file. Open a terminal and navigate to that directory. You can use the standard `npm install` command to install modules for your TypeScript reactors.

The important takeaway here is with the package.json file and the compiled JavaScript in the Foo/dist/ directory, you have a standard Node.js program that executes as such. You can modify and debug it just as you would a Node.js program.

Reactions

Recall that a reaction is defined within a reactor using the following syntax:

```
reaction(triggers) uses -> effects {=
  ... target language code ...
=}
```

In this section, we explain how **triggers**, **uses**, and **effects** variables work in the TypeScript target.

Types

In Lingua Franca, reactor elements like inputs, outputs, actions, parameters, and state are typed using target language types. For the TypeScript target, TypeScript types are generally acceptable with two notable exceptions:

- Custom types (and classes) must be defined in the preamble before they may be used.
- `undefined` is not a valid type for an input, output, or action. This is because `undefined` is used to designate the absence of an input, output, or action during a reaction.

To benefit from type checking, you should declare types for your reactor elements. If a type isn't declared for a state variable, it is assigned the type `unknown`. If a type isn't declared for an input, output, or action, it is assigned the reactor-ts type `Present` which is defined as

```
export type Present = (number | string | boolean | symbol | object | null);
```

Inputs and Outputs

In the body of a reaction in the TypeScript target, inputs are simply referred to by name. An input of type `t` is available within the body of a reaction as a local variable of type `t | undefined`. To determine whether an input is present, test the value of the input against `undefined`. An `undefined` input is not present.

WARNING Be sure to use the `===` or `!==` operator and not `==` or `!=` to test against `undefined`. In JavaScript/TypeScript the comparison `undefined == null` yields the value `true`. It may also be tempting to rely upon the falsy evaluation of `undefined` within an `if` statement, but this may introduce bugs. For example a reaction that tests the presence of input `x` with `if (x) { ... }` will not correctly identify potentially valid present values such as `0`, `false`, or `""`.

For example, the `Determinism.lf` test case in the test directory includes the following reactor:

```
reactor Destination {
    input x:number;
    input y:number;
    reaction(x, y) {=
        let sum = 0;
        if (x !== undefined) {
            sum += x;
        }
        if (y !== undefined) {
            sum += y;
        }
        console.log("Received " + sum);
        if (sum != 2) {
            console.log("FAILURE: Expected 2.");
            util.failure();
        }
    =}
}
```

The reaction refers to the inputs `x` and `y` by name and tests for their presence by testing `x` and `y` against `undefined`. If a reaction is triggered by just one input, then normally it is not necessary to test for its presence. It will always be present. However TypeScript's type system is not smart enough to know such an input will never have type `undefined` if there's no test against `undefined` within the reaction. An explicit type annotation (for example `x = x as t;` where `t` is the type of the input) may be necessary to avoid type errors from the compiler. In the above example, there are two triggers, so the reaction has no assurance that both will be present.

Inputs declared in the **uses** part of the reaction do not trigger the reaction. Consider this modification of the above reaction:

```
reaction(x) y {=
    let sum = x as number;
    if (y !== undefined) {
        sum += y;
    }
    console.log("Received " + sum + ".");
=}
```

It is no longer necessary to test for the presence of `x` because that is the only trigger. The input `y`, however, may or may not be present at the logical time that this reaction is triggered. Hence, the code must test for its presence.

The **effects** portion of the reaction specification can include outputs and actions. Actions will be described below. Like inputs, an output of type `t` is available within the body of a reaction as a local variable of type `t | undefined`. The local variable for each output is initialized to the output's current value. Outputs are set by assigning a (non-`undefined`) value to its local variable (no changes will be made to an output if it has the value `undefined` at the end of a reaction). Whatever value an output's local variable has at the end of the reaction will be set to that output. If an output's local variable has the value `undefined` at the end of the reaction, that output will not be set and connected downstream inputs will be absent. For example, we can further modify the above example as follows:

```
output z:number;
reaction(x) y -> z {=
    let sum = x as number;
    if (y !== undefined) {
        sum += y;
    }
    z = sum;
=}
```

If an output gets set more than once at any logical time, downstream reactors will see only the *final* value that is set. Since the order in which reactions of a reactor are invoked at a logical time is deterministic, and whether inputs are present depends only on their timestamps, the final value set for an output will also be deterministic.

An output may even be set in different reactions of the same reactor at the same logical time. In this case, one reaction may wish to test whether the previously invoked reaction has set the output. It can do that using a `!== undefined` test for that output. For example, the following reactor will always produce the output 42:

```
reactor TestForPreviousOutput {
    output out:number;
    reaction(startup) -> out {=
        if (Math.random() > 0.5) {
            out = 21;
        }
    =}
    reaction(startup) -> out {=
        let previous_output = out;
        if (previous_output) {
            out = 2 * previous_output;
        } else {
            out = 42;
        }
    =}
}
```

The first reaction may or may not set the output to 21. The second reaction doubles the output if it has been previously produced and otherwise produces 42.

Using State Variables

A reactor may declare state variables, which become properties of each instance of the reactor. For example, the following reactor will produce the output sequence 0, 1, 2, 3, ... :

```
reactor Count {
    state count:number(0);
    output y:number;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        count++;
        y = count;
    =}
}
```

The declaration on the second line gives the variable the name "count", declares its type to be `number`, and initializes its value to 0. The type and initial value can be enclosed in the Typescript-code delimitters `{= ... =}` if they are not simple identifiers, but in this case, that is not necessary.

In the body of the reaction, the reactor's state variable is referenced by way of a local variable of the same name. The local variable will contain the current value of the state at the beginning of the reaction. The final value of the local variable will be used to update the state at the end of the reaction.

It may be tempting to declare state variables in the **preamble**, as follows:

```
reactor FlawedCount {
    preamble {=
        let count = 0;
    =}
    output y:number;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        count++;
        y = count;
    =}
}
```

This will produce a sequence of integers, but if there is more than one instance of the reactor, those instances will share the same variable count. Hence, **don't do this**! Sharing variables across instances of reactors violates a basic principle, which is that reactors communicate only by sending messages to one another. Sharing variables will make your program nondeterministic. If you have multiple instances of the above FlawedCount reactor, the outputs produced by each instance will not be predictable, and in an asynchronous implementation, will also not be repeatable.

A state variable may be a time value, declared as follows:

```
    state time_value:time(100 msec);
```

The `time_value` variable will be of type `TimeValue`, which is an object used to represent a time in the TypeScript Target. Refer to the section on timed behavior for more information.

A state variable can have an array or object value. For example, the following reactor computes the **moving average** of the last four inputs each time it receives an input:

```
reactor MovingAverage {
    state delay_line:{=Array<number>=}({= [0.0, 0.0, 0.0] =});
    state index:number(0);
    input x:number;
    output out:number;
    reaction(x) -> out {=
        x = x as number;
        // Calculate the output.
        let sum = x;
        for (let i = 0; i < 3; i++) {
            sum += delay_line[i];
        }
        out = sum/4.0;

        // Insert the input in the delay line.
        delay_line[index] = x;

        // Update the index for the next input.
        index++;
        if (index >= 3) {
            index = 0;
        }
    =}
}
```

The second line declares that the type of the state variable is an array of `number`s with the initial value of the array being a three-element array filled with zeros.

States whose type are objects can similarly be initialized. Declarations can take an object literal as the initial value:

```
state myLiteral:{= {foo: number, bar: string} =}({= {foo: 42, bar: "baz"} =
```

or use `new`:

```
state mySet:{=Set<number>=}({= new Set<number>() =});
```

Using Parameters

Reactor parameters are also referenced in the TypeScript code as local variables. The example below modifies the above `Count` reactor so that its stride is a parameter:

```
target TypeScript;
reactor Count(stride:number(1)) {
    state count:number(0);
    output y:number;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        y = count;
        count += stride;
    =}
}
reactor Display {
    input x:number;
    reaction(x) {=
        console.log("Received: " + x + ".");
    =}
}
main reactor Stride {
    c = new Count(stride = 2);
    d = new Display();
    c.y -> d.x;
}
```

The second line defines the `stride` parameter, gives its type, and gives its initial value. As with state variables, the type and initial value can be enclosed in `{= ... =}` if necessary. The parameter is referenced in the reaction by referring to the local variable `stride`.

When the reactor is instantiated, the default parameter value can be overridden. This is done in the above example near the bottom with the line:

```
    c = new Count(stride = 2);
```

If there is more than one parameter, use a comma separated list of assignments.

Parameters in Lingua Franca are immutable. To encourage correct usage, parameter variables within a reaction are local `const` variables. If you feel tempted to use a mutable parameter, instead try using the parameter to initialize state and modify the state variable instead. This is illustrated below by a further modification to the Stride example where it takes an initial "start" value for count as a second parameter:

```
target TypeScript;
reactor Count(stride:number(1), start:number(5)) {
    state count:number(start);
    output y:number;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        y = count;
        count += stride;
    =}
}
reactor Display {
    input x:number;
    reaction(x) {=
        console.log("Received: " + x + ".");
    =}
}
main reactor Stride {
    c = new Count(stride = 2, start = 10);
    d = new Display();
    c.y -> d.x;
}
```

Parameters can have array or object values. Here is an example that outputs the elements of an array as a sequence of individual messages:

```
reactor Source(sequence:{=Array<number>=}({= [0, 1, 2] =})) {
    output out:number;
    state count:number(0);
    logical action next;
    reaction(startup, next) -> out, next {=
        out = sequence[count];
        count++;
        if (count < sequence.length) {
            actions.next.schedule(0, null);
        }
    =}
}
```

The **logical action** named `next` and the `schedule` function are explained below in Scheduling Delayed Reactions, but here they are used simply to repeat the reaction until all elements of the array have been sent.

Above, the parameter default value is an array with three elements, `[0, 1, 2]`. The syntax for giving this default value is a TypeScript array literal. Since this is TypeScript syntax, not Lingua Franca syntax, the initial value needs to be surrounded with the target code delimiters, `{= ... =}`. The default value can be overridden when instantiating the reactor using a similar syntax:

```
s = new Source(sequence={= [1, 2, 3, 4] =});
```

Both default and overridden values for parameters can also be created with the `new` keyword:

```
reactor Source(sequence:{=Array<number>=}({= new Array<number>() =})) {
```

and

```
s = new Source(sequence={= new Array<number() =});
```

Sending and Receiving Custom Types

You can define your own datatypes in TypeScript and send and receive those. Consider the following example:

```
reactor CustomType {
    preamble {=
        type custom = string | null;
    =}
    output out:custom;
    reaction(startup) -> out {=
        out = null;
    =}
}
```

The **preamble** code defines a custom union type of `string` and `null`.

Timed Behavior

See Summary of Time Functions and Utility Function Reference for a quick API reference.

Timers are specified exactly as in the Lingua Franca language specification. When working with time in the TypeScript code body of a reaction, however, you will need to know a bit about its internal representation.

A `TimeValue` is an class defined in the TypeScript target library file `time.ts` to represent a time instant or interval. For your convenience `TimeValue` and other classes from the `time.ts` library mentioned in these instructions are automatically imported into scope of your reactions. An instant

is the number of nanoseconds that have elapsed since January 1, 1970. An interval is the difference between two instants. When an LF program starts executing, logical time is (normally) set to the instant provided by the operating system. (On some embedded platforms without real-time clocks, it will be set instead to zero.)

Internally a `TimeValue` uses two numbers to represent the time. To prevent overflow (which would occur for time intervals spanning more than 0.29 years if a single JavaScript number, which has 2^53 bits of precision, were to be used), we use *two* numbers to store a time value. The first number denotes the number of whole seconds in the interval or instant; the second number denotes the remaining number of nanoseconds in the interval or instant. The first number represents the number of seconds, the second number represents the number of nanoseconds. These fields are not accessible to the programmer, instead `TimeValue`s may be manipulated by an API with functions for addition, subtraction, and comparison.

A reaction can examine the current logical time (which is constant during the execution of the reaction). For example, consider:

```
target TypeScript;
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        let logical:TimeValue = util.getCurrentLogicalTime()
        console.log("Logical time is " + logical + ".");
    =}
}
```

When executed, you will get something like this:

```
Logical time is (1584666585 secs; 805146880 nsecs).
Logical time is (1584666586 secs; 805146880 nsecs).
Logical time is (1584666587 secs; 805146880 nsecs).
...
```

Subsequent values of logical time are printed out in their raw form, of seconds and nanoseconds. If you look closely, you will see that each number is one second larger than the previous number.

You can also obtain the *elapsed* logical time since the start of execution, rather than exact logical time:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        let logical:TimeValue = util.getElapsedLogicalTime()
        console.log("Logical time is " + logical + ".");
    =}
}
```

This will produce:

```
Logical time is (0 secs; 0 nsecs).
Logical time is (1 secs; 0 nsecs).
Logical time is (2 secs; 0 nsecs).
...
```

You can get physical time, which comes from your platform's real-time clock:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        let physical:TimeValue = util.getCurrentPhysicalTime()
        console.log("Physical time is " + physical + ".");
    =}
}
```

This will produce something like this:

```
Physical time is (1584666801 secs; 644171008 nsecs).
Physical time is (1584666802 secs; 642269952 nsecs).
Physical time is (1584666803 secs; 642278912 nsecs).
...
```

Notice that these numbers are increasing by *roughly* one second each time.

You can also get *elapsed* physical time from the start of execution:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        let physical:TimeValue = util.getElapsedPhysicalTime()
        console.log("Physical time is " + physical + ".");
    =}
}
```

This will produce something like:

```
Physical time is (0 secs; 2260992 nsecs).
Physical time is (1 secs; 166912 nsecs).
Physical time is (2 secs; 136960 nsecs).
...
```

You can create a `TimeValue` yourself with the `UnitBasedTimeValue` class. `UnitBasedTimeValue` is a subclass of `TimeValue` and can be used wherever you could also use a `TimeValue` directly obtained from one of the `util` functions. A `UnitBasedTimeValue` is constructed with a whole number and a `TimeUnit`. A `TimeUnit` is an enum from the `time.ts` library with convenient labels for common time units. These are nsec, usec, msec, sec (or secs), minute (or minutes), hour (or hours), day (or days), and week (or weeks).

This reactor has an example of a UnitBasedTimeValue.

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        let myTimeValue:TimeValue = new UnitBasedTimeValue(200, TimeUnit.ms
        let logical:TimeValue = util.getCurrentLogicalTime()
        console.log("My custom time value is " + myTimeValue + ".");
    =}
```

This will produce:

```
My custom time value is 200 msec.
My custom time value is 200 msec.
My custom time value is 200 msec.
...
```

Tags

The TypeScript target provides a utility to get the current `Tag` of a reaction. Recall that time in Lingua Franca is superdense and each `TimeValue` is paired with an integer "microstep" index to track the number of iterations at a particular `TimeValue`. A `Tag` is this combination of a TimeValue and a "microstep". The `time.ts` library provides functions for adding, subtracting, and comparing `Tag`s.

You can get the current `Tag` in your reactions. This example illustrates tags with a Zero-Delay Action:

```
target TypeScript;
main reactor GetTime {
    timer t(0, 1 sec);
    logical action a;
    reaction(t) -> a {=
        let superdense:Tag = util.getCurrentTag();
        console.log("First iteration - the tag is: " + superdense + ".");
        actions.a.schedule(0, null);
    =}
    reaction(a) {=
        let superdense:Tag = util.getCurrentTag();
        let timePart:TimeValue = superdense.time;
        let microstepPart:number = superdense.microstep;
        console.log("Second iteration - the time part of the tag is:  " + t
        console.log("Second iteration - the microstep part of the tag is:
    =}
}
```

This will produce:

```
First iteration - the tag is: ((1584669987 secs; 740464896 nsecs), 0).
Second iteration - the time part of the tag is:  (1584669987 secs; 74046489
Second iteration - the microstep part of the tag is:  1.
First iteration - the tag is: ((1584669988 secs; 740464896 nsecs), 0).
Second iteration - the time part of the tag is:  (1584669988 secs; 74046489
Second iteration - the microstep part of the tag is:  1.
First iteration - the tag is: ((1584669989 secs; 740464896 nsecs), 0).
Second iteration - the time part of the tag is:  (1584669989 secs; 74046489
Second iteration - the microstep part of the tag is:  1.
...
```

The first reaction prints the "First iteration" part of the output at microstep 0. The second reaction occurs one microstep later (explained in Scheduling Delayed Reactions) and illustrates how to split a `Tag` into its constituent `TimeValue` and microstep.

Scheduling Delayed Reactions

Each action listed as an **effect** for a reaction is available as a schedulable object in the reaction body via the `actions` object. The TypeScript target provides a special `actions` object with a property for each schedulable action. Schedulable actions (of type `t`) have the object method:

```
schedule: (extraDelay: TimeValue | 0, value?: T) => void;
```

The first argument can either be the literal 0 (shorthand for 0 seconds) or a `TimeValue` / `UnitBasedTimeValue`. The second argument is the value for the action. Consider the following reactor:

```
target TypeScript;
reactor Schedule {
    input x:number;
    logical action a;
    reaction(x) -> a {=
        actions.a.schedule(new UnitBasedTimeValue(200, TimeUnit.msec), null
    =}
    reaction(a) {=
        let elapsedTime = util.getElapsedLogicalTime();
        console.log("Action triggered at logical time " + elapsedTime + " a
    =}
}
```

When this reactor receives an input `x`, it calls `schedule()` on the action `a`, so it will be triggered at the logical time offset (200 msec) with a null value. The action `a` will be triggered at a logical time 200 milliseconds after the arrival of input `x`. This will trigger the second reaction, which will use the `util.getElapsedLogicalTime()` function to determine how much logical time has elapsed since the start of execution. The third argument to the `schedule()` function is a **value**, data that can be carried by the action, which is explained below. In the above example, there is no value.

Zero-Delay Actions

If the specified delay in a `schedule()` call is zero, then the action `a` will be triggered one **microstep** later in **superdense time** (see Superdense Time). Hence, if the input `x` arrives at metric logical time *t*, and you call `schedule()` as follows:

```
actions.a.schedule(0);
```

then when a reaction to `a` is triggered, the input `x` will be absent (it was present at the *previous* microstep). The reaction to `x` and the reaction to `a` occur at the same metric time *t*, but separated by one microstep, so these two reactions are *not* logically simultaneous. These reactions execute with different Tags.

Actions With Values

If an action is declared with a data type, then it can carry a **value**, a data value that becomes available to any reaction triggered by the action. The most common use of this is to implement a logical delay, where a value provided at an input is produced on an output with a larger logical timestamp. To accomplish that, it is much easier to use the **after** keyword on a connection between reactors. Nevertheless, in this section, we explain how to directly use actions with value. In fact, the **after** keyword is implemented as described below.

If you are familiar with other targets (like C) you may notice it is much easier to schedule actions with values in TypeScript because of TypeScript/JavaScript's garbage collected memory management. The following example implements a logical delay using an action with a value.

```
reactor Delay(delay:time(100 msec)) {
    input x:number;
    output out:number;
    logical action a:number;
    reaction(x) -> a {=
        actions.a.schedule(delay, x as number);
    =}
    reaction(a) -> out {=
        if (a !== null){
            out = a as number
        }
    =}
}
```

The action `a` is specified with a type `number`. The first reaction declares `a` as its effect. This declaration makes it possible for the reaction to schedule a future triggering of `a`. It's necessary to explicitly annotate the type of `x` as a number in the schedule function because TypeScript doesn't know the only trigger of a reaction must be present in that reaction.

The second reaction declares that it is triggered by `a` and has effect `out`. When a reaction triggers or uses an action the value of that action is made available within the reaction as a local variable with the name of the action. This variable will take on the value of the action and it will have the value `undefined` if that action is absent because it was not scheduled for this reaction.

The local variable cannot be used directly to schedule an action. As described above, an action `a` can only be scheduled in a reaction when it is 1) declared as an effect and 2) referenced through a property of the `actions` object. The reason for this implementation is that `actions.a` refers to the **action**, not its value, and it is possible to use both the action and the value in the same reaction. For example, the following reaction will produce a counting sequence after it is triggered the first time:

```
reaction(a) -> out, a {=
    if (a !== null) {
        a = a as number;
        out = a;
        let newValue = a++;
        actions.a.schedule(delay, newValue);
    }
=}
```

Stopping Execution

A reaction may request that the execution stop by calling the function `util.requestShutdown()` which takes no arguments. Execution will not stop immediately when this function is called; all events with the current tag will finish processing and execution will continue for one more microstep to give shutdown triggers a chance to execute. After this additional step, execution will terminate.

Implementation Details

When a TypeScript reactor is compiled, the generated code is placed inside a project directory. This is because there are two steps of compilation. First, the Lingua Franca compiler generates a TypeScript project from the TypeScript reactor code. Second, the Lingua Franca compiler runs a TypeScript compiler on the generated TypeScript project to produce executable JavaScript. This is illustrated below:

```
Lingua Franca (.lf) ==> TypeScript (.ts) ==> JavaScript (.js)
```

Assuming the directory containing our Lingua Franca file `Foo.lf` is named `TS`, the compiler will generate the following:

1. TS/package.json
2. TS/node_modules
3. TS/Foo/tsconfig.json
4. TS/Foo/babel.config.js
5. TS/Foo/src/
6. TS/Foo/dist/

Items 1, 3, and 4 are configuration files for the generated project. Item 2 is a node_modules directory with contents specified by item 1. Item 5 is the directory for generated TypeScript code. Item 6 is the directory for compiled JavaScript code. In addition to the generated code for your Lingua Franca program, items 5 and 6 include libraries from the reactor-ts submodule.

The Lingua Franca compiler automatically invokes other programs as it compiles a Lingua Franca (.lf) file to a Node.js executable JavaScript (.js) file. The files package.json, babel.config.js, and tsconfig.json are used to configure the behavior of those other programs. Whenever you compile a .lf file for the first time, the Lingua Franca compiler will copy default versions of these configuration files into the new project so the other programs can run. **The Lingua Franca compiler will only copy a default configuration file into a project if that file is not already present in the generated project.** This means you, the reactor programmer, may safely modify these configuration files to control the finer points of compilation. Beware, other generated files in the project's src and dist directories may be overwritten by the compiler.

package.json

Node.js uses a package.json file to describe metadata relevant to a Node project. This includes a list of project dependencies (i.e. modules) used by the project. When the Lingua Franca compiler copies a default package.json file into a Lingua Franca project that doesn't already have a package.json, the compiler runs the command `npm install` to create a node_modules directory. The default package.json only lists dependencies for the reactor-ts submodule. Follow these instructions to modify package.json if you want to use other Node modules in your reactors.

tsconfig.json

After generating a TypeScript program from a .lf file, the Lingua Franca compiler uses the TypeScript compiler `tsc` to run a type check. The behavior of `tsc` is configured by the tsconfig.json file. You probably won't need to modify tsconfig.json, but you can if you know what you're doing.

babel.config.js

If the `tsc` type check was successful, the Lingua Franca compiler uses `babel` to compile the generated TypeScript code into JavaScript. (This blog post articulates the advantages of using `babel` over `tsc` to generate JavaScript.) There are many different flavors of JavaScript and the babel.config.js file specifies exactly what `babel` should generate. This is the file to edit if you want the Lingua Franca compiler to produce a different version of JavaScript as its final output.

Debugging Type Errors

Let's take the minimal reactor example, and intentionally break it by adding a type error into the reaction.

```
target TypeScript;
main reactor ReactionTypeError {
    timer t;
    reaction(t) {=
        let foo:number = "THIS IS NOT A NUMBER";
        console.log("Hello World.");
    =}
}
```

This reactor will not compile, and should you attempt to compile it you will get an output from the compiler which looks something like this:

```
--- Standard output from command:
src/ReactionTypeError.ts(23,25): error TS2322: Type '"THIS IS NOT A NUMBER"

--- End of standard output.
```

In particular the output

```
src/ReactionTypeError.ts(23,25): error TS2322: Type '"THIS IS NOT A NUMBER"
```

identifies the problem: surprisingly, the string `"THIS IS NOT A NUMBER"` is not a number. However the line information `(23,25)` is a little confusing because it points to the location of the type error **in the generated** .ts file `ReactionTypeError/src/ReactionTypeError.ts` not in the original .lf file `ReactionTypeError.lf`. The .ts files produced by the TypeScript code generator are quite readable if you are familiar with the reactor-ts submodule, but even if you aren't familiar it is not too difficult to track down the problem. Just open `ReactionTypeError/src/ReactionTypeError.ts` in your favorite text editor (we recommend Visual Studio for its excellent TypeScript integration) and look at line 23.

```
14        this.addReaction(
15            new Triggers(this.t),
16            new Args(this.t),
17            function (this, __t: Readable<Tag>) {
18                // =============== START react prologue
19                const util = this.util;
20                let t = __t.get();
21                // =============== END react prologue
22                try {
23                    let foo:number = "THIS IS NOT A NUMBER";
24                    console.log("Hello World.");
25                } finally {
26                    // =============== START react epilogue
27
28                    // =============== END react epilogue
29                }
30            }
31        );
```

There (inside the try block) we can find the problematic reaction code. *Reaction code is copied verbatim into generated .ts files*.

It can be a bit harder to interpret type errors outside of reaction code, but most type error messages are still relatively clear. For example if you attempt to connect a reactor output to an incompatibly typed input like:

```
target TypeScript;
main reactor ConnectionError {
    s = new Sender();
    r = new Receiver();
    s.foo -> r.bar;
}
reactor Sender {
    output foo:number;
}
reactor Receiver {
    input bar:string;
}
```

you should get an error like

```
--- Standard output from command:
src/InputTypeError.ts(36,23): error TS2345: Argument of type 'OutPort<numbe
  Types of property 'value' are incompatible.
    Type 'number | undefined' is not assignable to type 'string | undefined
      Type 'number' is not assignable to type 'string | undefined'.

--- End of standard output.
```

The key message being `Argument of type 'OutPort<number>' is not assignable to parameter of type 'Port<string>'`.

One last tip: if you attempt to reference a port, action, timer etc. named `foo` that isn't declared in the triggers, uses, or effects declaration of the reaction, you will get the error `Cannot find name 'foo'` in the reaction body.

Utility Function Reference

These utility functions may be called within a TypeScript reaction:

`util.requestShutdown(): void` Ends execution after one microstep. See Stopping Execution.

`util.getCurrentTag(): Tag` Gets the current (logical) tag. See Tags.

`util.getCurrentLogicalTime(): TimeValue` Gets the current logical TimeValue. See Time.

`util.getCurrentPhysicalTime(): TimeValue` Gets the current physical TimeValue. See Time.

`util.getElapsedLogicalTime(): TimeValue` Gets the elapsed logical TimeValue from execution start. See Time.

`util.getElapsedPhysicalTime(): TimeValue` Gets the elapsed physical TimeValue from execution start. See Time.

`util.success(): void` Invokes the reactor-ts App's default success callback. FIXME: Currently doesn't do anything in Lingua Franca.

`util.failure(): void` Invokes the reactor-ts App's default failure callback. Throws an error.

Summary of Time Functions

See Time. These time functions are defined in the time.ts library of reactor-ts.

`UnitBasedTimeValue(value: number, unit:TimeUnit)` Constructor for `UnitBasedTimeValue`, a programmer-friendly subclass of TimeValue. Use a number and a `TimeUnit` enum.

```
enum TimeUnit {
    nsec,
    usec,
    msec,
    sec,
    secs,
    minute,
    minutes,
    hour,
    hours,
    day,
    days,
    week,
    weeks
}
```

`TimeValue.add(other: TimeValue): TimeValue` Adds `this` to `other`.

`TimeValue.subtract(other: TimeValue): TimeValue` Subtracts `other` from `this`. A negative result is an error.

`TimeValue.difference(other: TimeValue): TimeValue` Obtain absolute value of `other` minus `this`.

`TimeValue.isEqualTo(other: TimeValue): boolean` Returns true if `this` and `other` represents the same TimeValue. Otherwise false.

`TimeValue.isZero(): boolean` Returns true if `this` represents a 0 TimeValue.

`TimeValue.isEarlierThan(other: TimeValue): boolean` Returns true if `this` < `other`. Otherwise false.

`Tag.isSmallerThan(other: Tag): boolean` Returns true if `this` < `other`. Otherwise false.

`Tag.isSimultaneousWith(other: Tag): boolean` Returns true if `this` and `other` represents the same Tag. Otherwise false.

`Tag.getLaterTag(delay: TimeValue): Tag` Returns a tag with the time part of this TimeValue incremented by delay.

`Tag.getMicroStepLater(): Tag` Returns a tag with the microstep part of this TimeValue incremented by 1.

`getTimeDifference(other: Tag): TimeValue` Returns a TimeValue that represents the absolute (i.e., positive) time difference between `this` and `other`.

Building Reactor-ts Documentation

FIXME: Host these docs somewhere.

To build and view proper documentation for `time.ts` (and other reactor-ts libraries), install typedoc and run

```
typedoc --out docs src
```

from the root of the reactor-ts. You probably already have the reactor-ts submodule at

```
lingua-franca/xtext/org.icyphy.linguafranca/src/lib/TS/reactor-ts/
```

You should see an output like.

```
Using TypeScript 3.8.3 from /usr/local/lib/node_modules/typescript/lib
Rendering [======================================] 100%

Documentation generated at /Users/<username>/git/lingua-franca/xtext/org.ic
```

Open that path in a browser with `/index.html` appended to the end like

```
/Users/<username>/git/lingua-franca/xtext/org.icyphy.linguafranca/src/lib/T
```

to navigate the docs.

Writing Reactors in Python

In the Python reactor target for Lingua Franca, reactions are written in Python. The user-written reactors are then generated into a Python 3 script that can be executed on several platforms. The Python target has been tested on Linux, MacOS, and Windows. To facilitate efficient and fast execution of Python code, the generated program relies on a C extension to facilitate Lingua Franca APIs such as `set` and `schedule`. To learn more about the structure of the generated Python program, see Implementation Details.

Python reactors can bring the vast library of scientific modules that exist for Python into a Lingua Franca program. Moreover, since the Python reactor target is based on a fast and efficient C runtime library, Lingua Franca programs can execute much faster than native equivalent Python programs in many cases. Finally, interoperability with C reactors is planned for the future.

> :spiral_notepad: In comparison to the C reactor target, the Python target can be up to an order of magnitude slower. However, depending on the type of application and the implementation optimizations in Python, you can achieve an on-par performance to the C target in many applications.

Setup

First, install Python 3 on your machine. See downloading Python.

> :spiral_notepad: The Python target requires a C implementation of Python (nicknamed CPython). This is what you will get if you use the above link, or with most of the alternative Python installations such as Anaconda. See this for more details.

The Python reactor target relies on `pip` and `setuptools` to be able to compile and install a Python C extension for each LF program. To install `pip3`, you can follow instructions here. `setuptools` can be installed using `pip3`:

```
pip3 install setuptools
```

> :spiral_notepad: A Python C extension is currently generated for each Lingua Franca program. To ensure cross-compatibility across multiple platforms, this extension is installed in the user space once code generation is finished (see Implementation Details). This extension module will have the name LinguaFranca[your_LF_program_name]. There is a handy script here that can uninstall all extension modules that are installed automatically by Lingua Franca tools (such as `lfc`).

A Minimal Example

A “Hello World” reactor for the target looks like this:

```
target Python;
main reactor Minimal {
    reaction(startup) {=
        print("Hello World.")
    =}
}
```

The `startup` trigger causes the reaction to execute at the logical start time of the program. This program can be found in a file called Minimal.lf in the [test directory], where you can also find quite a few more interesting examples. If you compile this using the `lfc` command-line compiler or the [Eclipse-base IDE], then a generated file called `Minimal.py` plus supporting files will be put into a subdirectory called `src-gen/Minimal`. If you are in the test directory, you can run the generated `Minimal.py` by running the following code in a shell:

```
python3 src-gen/Minimal/Minimal.py
```

The resulting output should look something like this:

```
---- Start execution at time Mon Oct 12 14:31:00 2020
---- plus 213090100 nanoseconds.
Hello World.
---- Elapsed logical time (in nsec): 0
---- Elapsed physical time (in nsec): 96,100
```

The Python Target Specification

To have Lingua Franca generate Python code, start your `.lf` file with the following target specification:

```
target Python;
```

A Python target specification may optionally specify any of the standard parameters (except for flags) that are supported by all targets.

For example, for the Python target, in a source file named `Foo.lf`, you might specify:

```
target Python {
    fast: true,
    timeout: 10 secs
};
```

The `fast` option given above specifies to execute the file as fast as possible, ignoring timing delays. This is achieved by not waiting for physical time to match logical time.

The `timeout` option specifies to stop after 10 seconds of logical time have elapsed.

These specify the *default* behavior of the generated code, the behavior it will exhibit if you give no command-line option. FIXME: command-line options are not supported yet.

> :warning: The LFC lexer does not support single-quoted strings in Python. This limitation also applies to target property values.

Command-Line Arguments

The Python reactor target currently does not support dynamically changing arguments at runtime.

Imports

The import statement can be used to share reactor definitions across several applications. Suppose for example that we modify the above Minimal.lf program as follows and store this in a file called HelloWorld.lf:

```
target Python;
reactor HelloWorld {
        state success(false);
        reaction(startup) {=
                print("Hello World.")
                self.success = True
        =}
}
main reactor HelloWorldTest {
        a = new HelloWorld();
}
```

This can be compiled and run, and its behavior will be identical to the version above. But now, this can be imported into another reactor definition as follows:

```
target Python;
import HelloWorld.lf;
main reactor TwoHelloWorlds {
        a = new HelloWorld();
        b = new HelloWorld();
}
```

This will create two instances of the HelloWorld reactor, and when executed, will print “Hello World” twice.

> :spiral_notepad: In the above example, the order in which the two reactions are invoked is undefined because there is no causal relationship between them.

A more interesting illustration of imports can be found in the Import.lf test case in the test directory.

Preamble

Reactions may contain arbitrary Python code, but often it is convenient for that code to use external packages and modules or to share class and method definitions. For either purpose, a reactor may include a preamble section. For example, the following reactor uses the `platform` module to print the platform information and a defined method to add 42 to an integer:

```
main reactor Preamble {
        preamble {=
                import platform
                def add_42(self, i):
                        return i + 42
        =}
        timer t;
        reaction(t) {=
                s = "42"
                i = int(s)
                print("Converted string {:s} to int {:d}.".format(s, i))
                print("42 plus 42 is ", self.add_42(42))
                print("Your platform is ", self.platform.system())
        =}
}
```

On a Linux machine, this will print:

```
Converted string 42 to int 42.
42 plus 42 is 84
Your platform is Linux
```

By putting import in the **preamble**, the module becomes available in all reactions of this reactor using the self modifier.

> :spiral_notepad: Preambles will be put in the generated Python class for the given reactor, and thus is part of the instance of the reactor (and cannot be shared between different instantiations of the reactor). For more information about implementation details of the Python target, see Implementation Details.

Alternatively, top level preambles could be used that don't belong to any particular reactor. These preambles can be used for functions such as import. The following example shows importing the hello module:

```
target Python {
    files: include/hello.py
};

preamble {=
import hello
=}
```

Notice the usage of the `files` target property to move the `hello.py` module located in the `include` folder of the test directory into the working directory (located in `src-gen/NAME`).

Reactions

Recall that a reaction is defined within a reactor using the following syntax:

> **reaction**(*triggers*) *uses* -> *effects* {=
> ... target language code ...
> =}

In this section, we explain how **triggers**, **uses**, and **effects** variables work in the Python target.

Types

In the Python target, reactor elements like inputs, outputs, actions, parameters, and state variables are not typed. This effectively allows for any valid Python object to be passed on these elements.

For more details and examples on using various Python object types, see Sending and Receiving Objects.

Inputs and Outputs

In the body of a reaction in the Python target, the value of an in put is obtained using the syntax `name.value`, where `name` is the name of the input port. To determine whether an input is present, use `name.is_present`. For example, the Determinism.lf test case in the test directory includes the following reactor:

```
reactor Destination {
    input x;
    input y;
    reaction(x, y) {=
        sm = 0
        if x.is_present:
            sm += x.value
        if y.is_present:
            sm += y.value
        print("Received ", sm)
        if sm != 2:
            sys.stderr.write("FAILURE: Expected 2.\n")
            exit(4)
    =}
}
```

The reaction refers to the input values `x.value` and `y.value` and tests for their presence by referring to the variables `x.is_present` and `y.is_present`. If a reaction is triggered by just one input, then normally it is not necessary to test for its presence; it will always be present. But in the above example, there are two triggers, so the reaction has no assurance that both will be present.

Inputs declared in the **uses** part of the reaction do not trigger the reaction. Consider the following modification to the above reaction:

```
reaction(x) y {=
    sm = x.value
    if y.is_present:
        sm += y.value;
    print("Received ", sm)
=}
```

It is no longer necessary to test for the presence of `x` because that is the only trigger. The input `y`, however, may or may not be present at the logical time that this reaction is triggered. Hence, the

code must test for its presence.

The **effects** portion of the reaction specification can include outputs and actions. Actions will be described below. Outputs are set using a `SET` macro. For example, we can further modify the above example as follows:

```
output z;
reaction(x) y -> z {=
    sm = x.value
    if y.is_present:
        sm += y.value
    z.set(sm)
=}
```

The `set` function on an output port will perform the following operation:

```
z.value = sm
z.is_present = True
```

The `set` function can be used to set any valid Python object. For more information, see Sending and Receiving Objects.

If an output gets set more than once at any logical time, downstream reactors will see only the *final* value that is set. Since the order in which reactions of a reactor are invoked at a logical time is deterministic, and whether inputs are present depends only on their timestamp, the final value set for an output will also be deterministic.

An output may even be set in different reactions of the same reactor at the same logical time. In this case, one reaction may wish to test whether the previously invoked reaction has set the output. It can check `name.is_present` to determine whether the output has been set. For example, the following reactor (see TestForPreviousOutput.lf) will always produce the output 42:

```
reactor Source {
    output out;
    preamble {=
        import random
    =}

    reaction(startup) -> out {=
        # Set a seed for random number generation based on the current time
        self.random.seed()
        # Randomly produce an output or not.
        if self.random.choice([0,1]) == 1:
            out.set(21)
    =}
    reaction(startup) -> out {=
        if out.is_present:
            out.set(2 * out.value)
        else:
            out.set(42)
    =}
}
```

The first reaction may or may not set the output to 21. The second reaction doubles the output if it has been previously produced and otherwise produces 42.

Using State Variables

A reactor may declare state variables, which become properties of each instance of the reactor. For example, the following reactor (see Count.lf) will produce the output sequence 1, 2, 3, ... :

```
reactor Count {
    state count(1);
    output out;
    timer t(0, 1 sec);
    reaction(t) -> out {=
        out.set(self.count)
        self.count += 1
    =}
}
```

The declaration on the second line gives the variable the name "count", and initializes its value to 1.

The initial value is the expression enclosed within the parentheses. It may be any LF expression, including an integer like seen above. LF supports only simple expression forms, if you need an

arbitrary Python expression, you can enclose it within the Python-code delimiters `{= ... =}` (see example below).

In the body of the reaction, the state variable is referenced using the syntax `self.count`. Here, `self` is a keyword that refers to the generated reactor class in Python and contains all the instance-specific data associated with an instance of the reactor. For more information regarding the implementation details of the Python target, see Implementation Details. Since each instance of a reactor has its own state variables, these variables are carried in the self object.

In certain cases, such as when more control is needed for initialization of certain class objects, this method might be preferable. Nonetheless, the code delimiters `{= ... =}` can also also be used. The following example, taken from StructAsState.lf demonstrates this usage:

```
main reactor StructAsState {
    preamble {=
        class hello:
            def __init__(self, name, value):
                self.name = name
                self.value = value
    =}
    state s ({=self.hello("Earth", 42) =});
    reaction(startup) {=
        print("State s.name='{:s}', value={:d}.".format(self.s.name, self.s
        if self.s.value != 42:
            sys.stderr.write("FAILED: Expected 42.\n")
            exit(1)
    =}
}
```

Notice that a class `hello` is defined in the preamble. The state variable `s` is then initialized to an instance of `hello` constructed within the `{= ... =}` delimiters.

State variables may be initialized to lists or tuples without requiring `{==}` delimiters. The following illustrates the difference:

```
target Python;
main reactor Foo {
    state a_tuple(1, 2, 3);
    state a_list([1, 2, 3]);
    reaction(startup) {=
        # will print "<class 'tuple'> != <class 'list'>"
        print("{0} != {1}".format(type(self.a_tuple), type(self.a_list)))
    =}
}
```

In Python, tuples are immutable, while lists can be modified. Be aware also that the syntax for declaring tuples in the Python target is the same syntax as to declare an array in the C target, so the immutability might be a surprise.

Using Parameters

Reactor parameters are also referenced in the Python code using the `self` object. The Stride.lf example modifies the above `Count` reactor so that its stride is a parameter:

```
target Python;
reactor Count(stride(1)) {
    state count(1);
    output y;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        y.set(self.count)
        self.count += self.stride
    =}
}
reactor Display {
    input x;
    state expected(1); // for testing.
    reaction(x) {=
        print("Received: ", x.value)
        if x.value != self.expected:
            sys.stderr.write("ERROR: Expected {:d}.\n".format(self.expected
        self.expected += 2
    =}
}
main reactor Stride {
    c = new Count(stride = 2);
    d = new Display();
    c.y -> d.x;
}
```

The second line defines the `stride` parameter and gives its initial value. As with state variables, types are not allowed. The initial value can be alternatively put in `{= ... =}` if necessary. The parameter is referenced in the reaction with the syntax `self.stride`.

When the reactor is instantiated, the default parameter value can be overridden. This is done in the above example near the bottom with the line:

```
c = new Count(stride = 2);
```

If there is more than one parameter, use a comma-separated list of assignments.

Like state variables, parameters can have list or tuple values. In the following example, the parameter `sequence` has as default value the list `[0, 1, 2]`:

```
reactor Source(sequence([0, 1, 2])) {
    output out;
    state count(0);
    logical action next;
    reaction(startup, next) -> out, next {=
        out.set(self.sequence[self.count])
        self.count+=1
        if self.count < len(self.sequence):
            next.schedule(0)
    =}
}
```

That default value can be overridden when instantiating the reactor using a similar syntax:

```
s = new Source(sequence = [1, 2, 3, 4]);
```

Notice that as any ordinary Python list, `len(self.sequence)` has been used in the code to deduce the length of the list.

In the above example, the **logical action** named `next` and the `schedule` function are explained below in Scheduling Delayed Reactions; here, they are used simply to repeat the reaction until all elements of the array have been sent.

Sending and Receiving Objects

You can define your own data types in Python and send and receive those. Consider the StructAsType example:

```
target Python {files: include/hello.py};

preamble {=
import hello
=}

reactor Source {
    output out;

    reaction(startup) -> out {=
        temp = hello.hello("Earth", 42)
        out.set(temp)
    =}
}
```

The top-level preamble has imported the hello module, which contains the following class:

```
class hello:
    def __init__(self, name = "", value = 0):
        self.name = name
        self.value = value
```

In the reaction to **startup**, the reactor has created an instance object of this class (as local variable named `temp`) and passed it downstream using the `set` method on output port `out`.

Alternatively, you can forego the variable and pass an instance object of the class directly to the port value, as is used in the StructAsTypeDirect example:

```
reactor Source {
    output out;
    reaction(startup) -> out {=
        out.value = hello.hello()
        out.value.name = "Earth"
        out.value.value = 42
        out.set(out.value)
    =}
}
```

The call to the `set` function is necessary to inform downstream reactors that the class object has a new value. In short, the `set` method is defined as follows:

> **.set***(value)*; Set the specified output (or input of a contained reactor) to the specified value. This value can be any Python object (including None and objects of type Any). The value is copied and therefore the variable carrying the value can be subsequently modified without changing the output.

A reactor receiving the class object message can take advantage of Python's duck typing and directly access the object:

```
reactor Print(expected(42)) {
    input _in;
    reaction(_in) {=
        print("Received: name = {:s}, value = {:d}\n".format(_in.value.name
                                                             _in.value.valu
    =}
}
```

> :spiral_notepad: The `hello` module has been imported using a top-level preamble, therefore, the contents of the module are available to all reactors defined in the current Lingua Franca file (similar

situation arises if the `hello` class itself was in the top-level preamble).

Timed Behavior

Timers are specified exactly as in the Lingua Franca language specification. When working with time in the Python code body of a reaction, however, you will need to know a bit about its internal representation.

In the Python target, similar to the C target, the value of a time instant or interval is an integer specifying a number of nanoseconds. An instant is the number of nanoseconds that have elapsed since January 1, 1970. An interval is the difference between two instants. When an LF program starts executing, logical time is (normally) set to the instant provided by the operating system (on some embedded platforms without real-time clocks, it will be set to zero instead).

Time in the Python target is an `int`, which is unbounded. For better clarity, two derived types are defined in `LinguaFrancaBase`, `instant_t` and `interval_t`, which you can use for time instants and intervals respectively. These are both equivalent to `int`, but using those types will insulate your code against changes and platform-specific customizations.

Lingua Franca uses a superdense model of time. A reaction is invoked at a logical **Tag**, an object consists of a `time` value (an int) and a `microstep` value (an unsigned int). The tag is guaranteed to not increase during the execution of a reaction. Outputs produced by a reaction have the same tag as the inputs, actions, or timers that trigger the reaction, and hence are **logically simultaneous**.

`Tag`s can be initialized using `Tag(time=some_number, microstep=some_other_number)`.

The functions for working with time are defined in pythontarget.c. The most useful functions are:

- `get_current_tag() -> Tag`: Returns a Tag instance of the current tag at which this reaction has been invoked.
- `get_logical_time() -> int`: Get the current logical time (the first part of the current tag).
- `get_microstep() -> unsigned int`: Get the current microstep (the second part of the current tag).
- `get_elapsed_logical_time() -> int`: Get the logical time elapsed since program start.
- `compare_tags(Tag, Tag) -> int`: Compare two `Tag` instances, returning -1, 0, or 1 for less than, equal, and greater than. `Tag`s can also be compared using rich comparators (ex. `<`, `>`, `==`), which returns `True` or `False`.

There are also some useful functions for accessing physical time:

- `get_physical_time() -> int`: Get the current physical time.
- `get_elapsed_physical_time() -> int`: Get the physical time elapsed since program start.
- `get_start_time() -> int`: Get the starting physical and logical time.

The last of these is both a physical and logical time because, at the start of execution, the starting logical time is set equal to the current physical time as measured by a local clock.

A reaction can examine the current logical time (which is constant during the execution of the reaction). For example, consider the GetTime.lf example:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        logical = get_logical_time()
        print("Logical time is ", logical)
    =}
}
```

When executed, you will get something like this:

```
---- Start execution at time Thu Nov  5 08:51:02 2020
---- plus 864237900 nanoseconds.
Logical time is  1604587862864237900
Logical time is  1604587863864237900
Logical time is  1604587864864237900
...
```

The first two lines give the current time-of-day provided by the execution platform at the start of execution. This is used to initialize logical time. Subsequent values of logical time are printed out in their raw form, rather than the friendlier form in the first two lines. If you look closely, you will see that each number is one second larger than the previous number, where one second is 1000000000 nanoseconds.

You can also obtain the *elapsed* logical time since the start of execution:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        elapsed = get_elapsed_logical_time()
        print("Elapsed logical time is ", elapsed)
    =}
}
```

This will produce:

```
---- Start execution at time Thu Nov  5 08:51:02 2020
---- plus 864237900 nanoseconds.
Elapsed logical time is  0
Elapsed logical time is  1000000000
Elapsed logical time is  2000000000
...
```

You can also get physical time, which comes from your platform's real-time clock:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        physical = get_physical_time()
        print("Physical time is ", physical)
    =}
}
```

This will produce something like this:

```
---- Start execution at time Thu Nov  5 08:51:02 2020
---- plus 864237900 nanoseconds.
Physical time is  1604587862864343500
Physical time is  1604587863864401900
Physical time is  1604587864864395200
...
```

Finally, you can get elapsed physical time:

```
main reactor GetTime {
    timer t(0, 1 sec);
    reaction(t) {=
        elapsed_physical = get_elapsed_physical_time()
        print("Elapsed physical time is ", elapsed_physical)
    =}
}
```

This will produce something like this:

```
---- Start execution at time Thu Nov  5 08:51:02 2020
---- plus 864237900 nanoseconds.
Elapsed physical time is  110200
Elapsed physical time is  1000185400
Elapsed physical time is  2000178600
...
```

Notice that these numbers are increasing by roughly one second each time. If you set the `fast` target parameter to `true`, then physical time will elapse much faster than logical time.

Working with nanoseconds in the Python code can be tedious if you are interested in longer durations. For convenience, a set of functions are available to the Python programmer to convert time units into the required nanoseconds. For example, you can specify 200 msec in Python code as `MSEC(200)` or two weeks as `WEEKS(2)`. The provided functions are `NSEC`, `USEC` (for microseconds), `MSEC`, `SEC`, `MINUTE`, `HOUR`, `DAY`, and `WEEK`. You may also use the plural of any of these. Examples are given in the next section.

Scheduling Delayed Reactions

The Python target provides a `.schedule()` method to trigger an action at a future logical time. Actions are described in the Language Specification document. Consider the Schedule reactor:

```
target Python;
reactor Schedule {
    input x;
    logical action a;
    reaction(a) {=
        elapsed_time = get_elapsed_logical_time()
        print("Action triggered at logical time {:d} nsec after start.".for
    =}
    reaction(x) -> a {=
        a.schedule(MSEC(200))
    =}
}
```

When this reactor receives an input `x`, it calls `a.schedule()`, specifying the action `a` to be triggered and the logical time offset (200 msec). The action `a` will be triggered at a logical time 200 milliseconds after the arrival of input `x`. At that logical time, the second reaction will trigger and will use the `get_elapsed_logical_time()` function to determine how much logical time has elapsed since the start of execution.

Notice that after the logical time offset of 200 msec, there may be another input `x` simultaneous with the action `a`. Because the reaction to `a` is given first, it will execute first. This becomes important when such a reactor is put into a feedback loop (see below).

Zero-Delay actions

If the specified delay in a `.schedule()` call is zero, then the action `a` will be triggered one **microstep** later in **superdense time** (see Superdense Time). Hence, if the input `x` arrives at metric

logical time t, and you call `.schedule()` as follows:

```
a.schedule(0)
```

then when a reaction to `a` is triggered, the input `x` will be absent (it was present at the *previous* microstep). The reaction to `x` and the reaction to `a` occur at the same metric time *t*, but separated by one microstep, so these two reactions are *not* logically simultaneous.

The metric time is visible to the Python programmer and can be obtained in a reaction using either `get_elapsed_logical_time()`, as above or `get_logical_time()`. The latter function also returns an `int` (aka `instant_t`), but its meaning is now the time elapsed since January 1, 1970 in nanoseconds.

As described in the Language Specification document, action declarations can have a *min_delay* parameter. This modifies the timestamp further. Also, the action declaration may be **physical** rather than **logical**, in which case the assigned timestap will depend on the physical clock of the executing platform.

Actions With Values

Actions can also carry a **value**, a Python object that becomes available to any reaction triggered by the action. This is particularly useful for physical actions that are externally triggered because it enables the action to convey information to the reactor. This could be, for example, the body of an incoming network message or a numerical reading from a sensor.

Recall from the Contained Reactors section in the Language Specification document that the **after** keyword on a connection between ports introduces a logical delay. This is actually implemented using a logical action. We illustrate how this is done using the DelayInt example:

```
reactor Delay(delay(100 msec)) {
    input _in;
    output out;
    logical action a;
    reaction(a) -> out {=
        if (a.value is not None) and a.is_present:
            out.set(a.value)
    =}
    reaction(_in) -> a {=
        a.schedule(self.delay, _in.value)
    =}
}
```

Using this reactor as follows:

```
d = new Delay();
source.out -> d._in;
d._in -> sink.out;
```

is equivalent to:

```
source.out -> sink.in after 100 msec;
```

The reaction to the input `in` declares as its effect the action `a`. This declaration makes it possible for the reaction to schedule a future triggering of `a`. As with other constructs in the Python reactor target, types are avoided.

The first reaction declares that it is triggered by `a` and has effect `out`. To read the value, it uses the `a.value` class variable. Because this reaction is first, the `out` at any logical time can be produced before the input `_in` is even known to be present. Hence, this reactor can be used in a feedback loop, where `out` triggers a downstream reactor to send a message back to `_in` of this same reactor. If the reactions were given in the opposite order, there would be causality loop and compilation would fail.

Stopping Execution

A reaction may request that the execution stop after all events with the current timestamp have been processed by calling the built-in function `request_stop()`, which takes no arguments. In a non-federated execution, the actual last tag of the program will be one microstep later than the tag at which `request_stop()` was called. For example, if the current tag is `(2 seconds, 0)`, the last (stop) tag will be `(2 seconds, 1)`.

> :spiral_notepad: The [[timeout | Target-Specification#timeout]] target specification will take precedence over this function. For example, if a program has a timeout of `2 seconds` and `request_stop()` is called at the `(2 seconds, 0)` tag, the last tag will still be `(2 seconds, 0)`.

Log and Debug Information

The Python supports the [[logging | Target-Specification#logging]] target specification. This will cause the runtime to produce more or less information about the execution. However, user-facing functions for different logging levels are not yet implemented (see issue #619).

Implementation Details

The Python target is built on top of the C runtime to enable maximum efficiency where possible. The Python target uses the [[single threaded C runtime | Writing-Reactors-in-C#single-threaded-implementation]] by default but will switch to the [[multithreaded C runtime | Writing-Reactors-in-C#multithreaded-implementation]] if a physical action is detected. The [[threads | Writing-Reactors-in-C#threads]] target specification can be used to override this behavior.

Running [[lfc | downloading-and-building#Command-Line-Tools]] on a `XXX.lf` program that uses the Python target specification will create the following files:

```
├── src
│   └── XXX.lf
└── src-gen
    └── XXX
        ├── core
        │   ...              # C runtime files
        ├── ctarget.c        # C target API implementations
        ├── ctarget.h        # C target API definitions
        ├── pythontarget.c   # Python target API implementations
        ├── pythontarget.h   # Python target API definitions
        ├── setup.py         # Setup file used to install the Python C exten
        ├── XXX.c            # Source code of the Python C extension
        └── XXX.py           # Actual Python code containing reactors and re
```

There are two major components in the `src-gen/XXX` directory that together enable the execution of a Python target application: A [[XXX.py | Writing-Reactors-in-Python#the-xxxpy-file-containing-user-code]] file containing the user code (e.g., reactor definitions and reactions) and the source code for a [[Python C extension module | Writing-Reactors-in-Python#the-generated-linguafrancaxxx-python-module-a-c-extension-module]] called `LinguaFrancaXXX` containing the C runtime, as well as hooks to execute the user-defined reactions. The interactions between the `src-gen/XXX/XXX.py` file and the `LinguaFrancaXXX` module are explained [[here | Writing-Reactors-in-Python#interactions-between-xxxpy-and-linguafrancaxxx]].

The `XXX.py` file containing user code

The `XXX.py` file contains all the reactor definitions in the form of Python classes. The contents of a reactor are converted as follows:

- Each **Reaction** in a reactor definition will be converted to a class method.
- Each **Parameter** will be converted to a class property to make it read-only.
- Each **State** variable will be converted to an instance variable.

- Each trigger and effect will be converted to an object passed as a method function argument to reaction methods, allowing the body of the reaction to access them.
- Each reactor **Preamble** will be put in the class definition verbatim.

Finally, each reactor class instantiation will be converted to a Python object class instantiation.

For example, imagine the following program:

```
# src/XXX.lf
target Python;
reactor Foo(bar(0)) {
    preamble {=
        import random
    =}
    state baz
    input _in
    logical action act
    reaction(_in, act) {=
        # Body of the reaction
        self.random.seed() # Note the use of self
    =}
}
main reactor {
    foo = new Foo()
}
```

Th reactor `Foo` and its instance, `foo`, will be converted to

```
# src-gen/XXX/XXX.py
...

# Python class for reactor Foo
class _Foo:

    # From the preamble, verbatim:
    import random
    def __init__(self, **kwargs):
        #Define parameters and their default values
        self._bar = 0
        # Handle parameters that are set in instantiation
        self.__dict__.update(kwargs)

        # Define state variables
        self.baz = None

    @property
    def bar(self):
        return self._bar

    def reaction_function_0(self , _in, act):
        # Body of the reaction
        self.random.seed() # Note the use of self
        return 0


# Instantiate classes
xxx_foo_lf = \
    [_Foo(bank_index = 0, \
        _bar=0)]

...
```

The generated LinguaFrancaXXX Python module (a C extension module)

The rest of the files in `src-gen/XXX` form a Python C extension module called `LinguaFrancaXXX` that can be installed by executing `python3 -m pip install .` in the `src-gen/XXX/` folder. In this case, `pip` will read the instructions in the `src-gen/XXX/setup.py` file and install a `LinguaFrancaXXX` module in your local Python module installation directory.

> :spiral_notepad: LinguaFrancaXXX does not necessarily have to be installed if you are using the "traditional" Python implementation (CPython) directly. You could simply use `python3 setup.py build` to build the module in the `src-gen/XXX` folder. However, we have found that other C Python implementations such as Anaconda will not work with this kind of local module.

As mentioned before, the LinguaFrancaXXX module is separate from `src-gen/XXX/XXX.py` but interacts with it. Next, we explain this interaction.

Interactions between XXX.py and LinguaFrancaXXX

The LinguaFrancaXXX module is imported in `src-gen/XXX/XXX.py`:

```
from LinguaFrancaXXX import *
```

This is done to enable the main function in `src-gen/XXX/XXX.py` to make a call to the `start()` function, which is part of the generated (and installed) `LinguaFrancaXXX` module. This function will start the main event handling loop of the C runtime.

From then on, `LinguaFrancaXXX` will call reactions that are defined in `src-gen/XXX/XXX.py` when needed.

The LinguaFrancaBase package

LinguaFrancaBase is a package that contains several helper methods and definitions that are necessary for the Python target to work. This module is installable via `python3 -m pip install LinguaFrancaBase` but is automatically installed if needed during the installation of `LinguaFrancaXXX`. The source code of this package can be found here.

This package's modules are imported in the `XXX.py` program:

```
from LinguaFrancaBase.constants import * #Useful constants
from LinguaFrancaBase.functions import * #Useful helper functions
from LinguaFrancaBase.classes import * #Useful classes
```

Already imported Python modules

The following packages are already imported and thus do not need to be re-imported by the user:

```
import sys
import copy
```

Examples

To see a few interactive examples written using the Python target, see here.

The Python CI tests might also act as a reference in some cases for the capabilities of the Python target.

Regression Tests

Regression Tests

Lingua Franca comes with an extensive set of regression tests that are executed on various platforms automatically whenever an update is pushed to the LF repository. There are three categories of tests:

- **System tests** are complete Lingua Franca programs that are compiled and executed automatically. A test passes if it successfully compiles and runs to completion with normal termination (return code 0). These tests are located in the `test` directory at the root of the LF repo, with one subdirectory per target language.
- **Unit tests** are Java or Xtend methods in the `compiler` package that are labeled with the `@test` directive in the comment just before the method. These tests check individual functions of the code generation infrastructure and/or IDE. The system tests are also executed through JUnit using methods with `@test` directives, but these tests are located in the `runtime` package (because their goal is not only to successfully generate code, but also to execute it and test for expected results).
- **Examples** are complete Lingua Franca programs that are compiled but not executed automatically. These are not executed automatically because they are typically interactive and may not terminate.

To learn about Lingua Franca, browsing the system tests and examples could be very useful.

Browsing and Editing Examples in the LF IDE

- In the LF IDE Eclipse, select File->New->Project (a General Project is adequate) and click Next.
- Give the project a name, e.g. `CExamples`.
- Unselect "Use default location".
- Navigate to the example directory and click Open. For the C target, the example directory is `$LF/example/C` where `$LF` is the root of your git clone of the lingua-franca repo.
- Click Finish.
- Open any one of the `.lf` files in any of the subdirectories.
- **IMPORTANT:** A dialog appears: Do you want to convert 'test' to an Xtext Project? Say YES.

This will create two directories called `src-gen` and `bin` in the `$LF/example/C` directory. Eclipse populates these directories with generated and compiled code. You can run the programs in the `bin` directory.

Browsing and Editing System Tests in the LF IDE

The built-in regression tests provide many small, simple examples of LF programs. Browsing them can be useful to learn the capabilities, so we strongly recommend creating a project for this.

- In the LF IDE Eclipse, select File->New->Project (a General Project is adequate) and click Next.
- Give the project a name, e.g. `CTests`.
- Unselect "Use default location".
- Navigate to the test directory and click Open. For the C target, the test directory is `$LF/test/C` where `$LF` is the root of your git clone of the lingua-franca repo.
- The `src` directory contains all the tests, some organized into topical subdirectories.
- Open any one of the `.lf` files.
- **IMPORTANT:** A dialog appears: Do you want to convert 'test' to an Xtext Project? Say YES.

This will create two directories called `src-gen` and `bin` in the `$LF/test/C` directory. Eclipse populates these directories with generated and compiled code. You can run the programs in the `bin` directory.

Running the Tests From the Command Line

The simplest way to run the regression tests is to use a Bash script called `run-lf-tests` in `$LF/bin`, which takes the target language as a parameter:

```
run-lf-tests C
run-lf-tests Cpp
run-lf-tests Python
run-lf-tests TS
```

This will run the system tests only. To run all the tests, use the `gradle` build system in the `$LF/xtext` directory:

```
cd $LF/xtext
./gradlew test
```

You can also selectively run just some of the tests. For example, to run the system tests for an individual target language, do this:

```
cd $LF
./gradlew test --tests org.lflang.tests.runtime.CTest.*
./gradlew test --tests org.lflang.tests.runtime.CppTest.*
./gradlew test --tests org.lflang.tests.runtime.PythonTest.*
./gradlew test --tests org.lflang.tests.runtime.TypeScriptTest.*
```

To run a single test case, use the `runSingleTest` gradle task along with the path to the test source file:

```
./gradlew runSingleTest --args test/C/src/Minimal.lf
```

It is also possible to run a subset of the tests. For example, the C tests are organized into the following categories:

- **generic** tests are `.lf` files located in `$LF/test/C/src`.
- **concurrent** tests are `.lf` files located in `$LF/test/C/src/concurrent`.
- **federated** tests are `.lf` files located in `$LF/test/C/src/federated`.
- **multiport** tests are `.lf` files located in `$LF/test/C/src/multiport`.

To invoke only the tests in the `concurrent` category, for example, do this:

```
cd $LF/xtext
./gradlew test --tests org.icyphy.tests.runtime.CTest.runConcurrentTests
```

Reporting Bugs

If you encounter a bug or add some enhancement to Lingua Franca, then you should create a regression test either as a system test or a unit test and issue a pull request. System tests are particularly easy to create since they are simply Lingua Franca programs that either compile and execute successfully (the test passes) or fail either to compile or execute.

Testing Architecture

System tests can be put in any subdirectory of `$LF/test` or `$LF/example`. Any `.lf` file within these directories will be treated as a system test unless they are within a directory named `failing`, in which case they will be ignored. The tests are automatically indexed by our JUnit-based test infrastructure, which is located in the package `xtext/org.icyphy.linguafranca.tests`. Each target has its own class in the `runtime` package, with a number of test methods that correspond to particular test categories, such as `generic`, `concurrent`, `federated`, etc. A test can be associated with a particular category by placing it in a directory that matches its name. For instance, we can create a test (e.g., `Foo.lf`) in

`test/C/src/concurrent`, which will then get indexed under the target `C` in the category `concurrent`. Files placed directly in `test/C/src` will be considered `generic` `C` tests, and a file in a directory `concurrent/federated` will be indexed as `federated` (corresponding to the nearest containing directory).

Caution: adding a *new* category requires updating an enum in `TestRegistry.java` and adding a `@test`-labeled method to `TestBase`.

Known Failures

Sometimes it is useful to retain tests that have a known failure that should be addressed at a later point. Such tests can simply be put in a directory called `failing`, which will tell our test indexing code to exclude it.

Examples

All files in our `example` directory are also indexed automatically. This is to assure they will continue to function correctly as our compiler evolves. All files in the `example` category will be parsed and code generated, but only files in a `test` directory will be executed, meaning they must also execute successfully in order not to be reported as a failure.

Test Output

Tests are grouped by target and category. It is also reported when, for a given category, there are other targets that feature tests that are missing for the target under test. Tests that either do not have a main reactor or are marked as known failures are reported as "ignored." For all the tests that were successfully indexed, it is reported how many passed. For each failing test, diagnostics are reported that should help explain the failure. Here is some sample output for `Ctest.runConcurrentTests`, which runs tests categorized as `concurrent` for the `C` target:

```
CTest > runConcurrentTests() STANDARD_OUT
    ================================================================================
    Target: C
    Description: Run concurrent tests.
    ================================================================================

    ================================================================================
    Category: CONCURRENT
    ================================================================================
    --------------------------------------------------------------------------------
    Ignored: 0
    --------------------------------------------------------------------------------

    --------------------------------------------------------------------------------
    Covered: 29/33
    --------------------------------------------------------------------------------
    Missing: src/concurrent/BankToBank.lf
    Missing: src/concurrent/BankToBankMultiport.lf
    Missing: src/concurrent/BankToBankMultiportAfter.lf
    Missing: src/concurrent/BankToMultiport.lf

    --------------------------------------------------------------------------------
    Passing: 29/29
    --------------------------------------------------------------------------------
```

Running JUnit Tests from Eclipse

It is also possible to invoke tests in Eclipse. Simply navigate to the source file in the package explorer, right click on it and select `Run As JUnit Test`. To invoke particular tests, open the file, right click on a method labeled as `@test` and again run it as a JUnit test.

Unit Tests

We also maintain a set of unit tests that focus on various aspects of the LF compiler. They are located in the `compiler` package in `xtext/org.icyphy.linguafranca.tests`. These tests can also be invoked from Eclipse as describe above, or from the command line as follows:

```
./gradlew test --tests org.icyphy.tests.compiler.*
```

Code Coverage

Code coverage is automatically recorded when running tests. After completing a test run, a full report can be found in `$LF/xtext/org.icyphy.linguafranca.tests/build/reports/html/jacoco/index.html`. Note that this report will only reflect the coverage of the test that have actually executed. It is possible to obtain the full report without waiting for all the tests to complete by running the following command which only parses and generates code for system tests (instead of building and executing them, too):

```
./gradlew test --tests org.icyphy.tests.runtime.compiler.CodeGenCoverage.*
```

Continuous Integration

Each push or pull request will trigger all tests to be run on Github Actions. It's configuration can be found here.

Contributing

Pull Requests

The preferred way to contribute to Lingua Franca is to set up Eclipse and the issue pull requests through GitHub.

Code Formatting

The Lingua Franca compiler builds on the Xtext framework and is written in Java and Xtend. A more elaborate description of our code formatting conventions will follow soon, but for those who use Eclipse, readily importable code formatters are available:

- Java
- Xtend