This copy of the Lingua Franca handbook for the cpp target was created on Friday, September 1, 2023 against commit a7067e.

Table of Contents

A First Reactor

See the requirements for using this target.

Minimal Example

A minimal but complete Lingua Franca file with one reactor is this:

```
target Cpp;

main reactor {
    reaction(startup) {=
        std::cout << "Hello World." << std::endl;
    =}
}
```

Every Lingua Franca program begins with a target declaration that specifies the language in which reactions are written. This is also the language of the program(s) generated by the Lingua Franca code generator.

Every LF program also has a **main** reactor, which is the top level of a hierarchy of contained and interconnected reactors. The above simple example has no contained reactors.

The **main** reactor above has a single **reaction**, which is triggered by the **startup** trigger. This trigger causes the reaction to execute at the start of the program. The body of the reaction, delimited by `{= ... =}`, is ordinary Cpp code which, as we will see, has access to a number of functions and variables specific to Lingua Franca.

Examples

Examples of Lingua Franca programs can be found in the examples-lingua-franca repository.

The regression tests have a rich set of examples that illustrate every feature of the language.

Structure of an LF Project

The Lingua Franca tools assume that LF programs are put into a file with a `.lf` extension that is stored somewhere within a directory called `src`. To compile and run the above example, choose

a **project root** directory, create a `src` directory within that, and put the above code into a file called, say, `src/HelloWorld.lf`. You can compile the code on the command line, within Visual Studio Code, or within the Epoch IDE. On the command line this will look like this:

```
> lfc src/HelloWorld.lf
... output from the code generator and compiler ...
```

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 (e.g., printing a message, as in the above `HelloWorld` example).

The general structure of a reactor definition is as follows:

Contents within square brackets are optional, contents within `<...>` are user-defined, and each line may appear zero or more times, as explained in the next pages. Parameters, 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.

Any number of reactors may be defined in one file, and a **main** reactor need not be given a name, but if it is given a name, then that name must match the file name.

Reactors may extend other reactors, inheriting their properties, and a file may import reactors from other files. If an imported LF file contains a **main** reactor, that reactor is ignored (it will not be 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.

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.
```

Inputs and Outputs

In this section, we will endow reactors with inputs and outputs.

Input and Output Declarations

Input and output declarations have the form:

For example, the following reactor doubles its input and sends the result to the output:

```
target Cpp;

reactor Double {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        if (x.is_present()){
            y.set(*x.get() * 2);
        }
    =}
}
```

Notice how the input value is accessed and how the output value is set. This is done differently for each target language. See the Target Language Details for detailed documentation of these mechanisms. Setting an output within a reaction will trigger downstream reactions at the same Logical Time that the reaction is invoked (or, more precisely, at the same tag). If a particular output port is set more than once at any tag, the last set value will be the one that downstream reactions see. 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.

The **reaction** declaration above indicates that an input event on port `x` is a **trigger** and that an output event on port `y` is a (potential) **effect**. A reaction can declare more than one trigger or effect by just listing them separated by commas. For example, the following reactor has two triggers and tests each input for presence before using it:

```
target Cpp;

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;
    =}
}
```

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 target dependent. In the C++ target, a smart pointer is returned for present values and `nullptr` if the value is not present.

Triggers, Effects, and Uses

The general form of a **reaction** is

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

The **triggers** field can be a comma-separated list of input ports, output ports of contained reactors, timers, actions, or the special events **startup** and **shutdown**. There must be at least one trigger for each reaction. A reaction with a **startup** trigger is invoked when the program begins executing, and a reaction with a **shutdown** trigger is invoked at the end of execution.

The **uses** field, which is optional, specifies input ports (or output ports of contained reactors) that do not trigger execution of the reaction but may be read by the reaction.

The **effects** field, which is also optional, is a comma-separated lists of output ports ports, input ports of contained reactors, or actions.

Setting an Output Multiple Times

If one or more reactions set an output multiple times at the same tag, then only the last value set will be seen by any downstream reactors.

If a reaction wishes to test whether an output has been previously set at the current tag by some other reaction, it can test it in the same way it tests inputs for presence.

Mutable Inputs

Normally, a reaction does not modify the value of an input. An input is said to be **immutable**. The degree to which this is enforced varies by target language. Most of the target languages make it rather difficult to enforce, so the programmer needs to avoid modifying the input. Modifying an input value may lead to nondeterministic results.

Occasionally, it is useful to modify an input. For example, the input may be a large data structure, and a reaction may wish to make a small modification and forward the result to an output. To accomplish this, the programmer should declare the input **mutable** as follows:

mutable input <name>;

This is a directive to the code generator indicating that reactions that read this input may also modify the value of the input. The code generator will attempt to optimize the scheduling to avoid copying the input value, but this may not be possible, in which case it will automatically insert a copy operation, making it safe to modify the input. The target-specific reference documentation has more details about how this works.

Parameters and State Variables

Parameter Declaration

A reactor class definition can parameterized as follows:

- A pointer type, such as `int*`.

Each parameter must have a *default value*, written `(<expr>)`. An expression may be a numeric constant, a string enclosed in quotation marks, a time value such as `10 msec`, a list of values, or target-language code enclosed in `{= ... =}`, for example. See Expressions for full details on what expressions are valid.

For example, the `Double` reactor on the previous page can be replaced with a more general parameterized reactor `Scale` as follows:

```
target Cpp;

reactor Scale(factor:int(2)) {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        y.set(factor * *x.get());
    =}
}
```

This reactor, given any input event `x` will produce an output `y` with value equal to the input scaled by the `factor` parameter. The default value of the `factor` parameter is 2, but this can be changed when the `Scale` reactor is instantiated.

Notice how, within the body of a reaction, the code accesses the parameter value. This is different for each target language.

State Declaration

A reactor declares a state variable as follows:

The `<value>` is an initial value and, like parameter values, can be given as an expression or target language code with delimiters `{= ... =}`. The initial value can also be given as a

parameter name. The value can be accessed and modified in a target-language-dependent way as illustrated by the following example:

```
target Cpp;

reactor Count {
    state count:int(0);
    output y:int;
    timer t(0, 100ms);

    reaction(t) -> y {=
        y.set(count++);
    =}
}
```

This reactor has an integer state variable named `count`, and each time its reaction is invoked, it outputs the value of that state variable and increments it. The reaction is triggered by a **timer**, discussed in the next section.

Time and Timers

Logical Time

A key property of Lingua Franca is **logical time**. All events occur at an instant in logical time. By default, the runtime system does its best to align logical time with **physical time**, which is some measurement of time on the execution platform. The **lag** is defined to be physical time minus logical time, and the goal of the runtime system is maintain a small non-negative lag.

The **lag** is allowed to go negative only if the `fast` target property or the --fast is set to `true`. In that case, the program will execute as fast as possible with no regard to physical time.

Time Values

A time value is given with units (unless the value is 0, in which case the units can be omitted). The allowable units are:

- For nanoseconds: `ns`, `nsec`, or `nsecs`
- For microseconds: `us`, `usec`, or `usecs`
- For milliseconds: `ms`, `msec`, or `msecs`
- For seconds: `s`, `sec`, `secs`, `second`, or `seconds`
- For minutes: `min`, `minute`, `mins`, or `minutes`
- For hours: `h`, `hour`, or `hours`
- For days: `d`, `day`, or `days`
- For weeks: `week` or `weeks`

The following example illustrates using time values for parameters and state variables:

```
target Cpp;

main reactor SlowingClock(start:time(100 msec), incr:time(100 msec)) {
    state interval:time(start);
    logical action a;
    reaction(startup) -> a {=
        a.schedule(start);
    =}

    reaction(a) -> a {=
        auto elapsed_logical_time = get_elapsed_logical_time();
        std::cout << "Logical time since start: " << elapsed_logical_time <
        interval += incr;
        a.schedule(interval);
    =}
}
```

This has two time parameters, `start` and `incr`, each with default value `100 msec`. This parameter is used to initialize the `interval` state variable, which also stores a time. The **logical action** `a`, explained below, is used to schedule events to occur at time `start` after program startup and then at intervals that are increased each time by `incr`. The result of executing this program will look like this:

```
Logical time since start: 100000000 nsec.
Logical time since start: 300000000 nsec.
Logical time since start: 600000000 nsec.
Logical time since start: 1000000000 nsec.
...
```

Timers

The simplest use of logical time in Lingua Franca is to invoke a reaction periodically. This is done by first declaring a **timer** using this syntax:

```
    timer <name>(<offset>, <period>);
```

The `<period>`, which is optional, specifies the time interval between timer events. The `<offset>`, which is also optional, specifies the (logical) time interval between when the program starts executing and the first timer event. If no period is given, then the timer event occurs only once. If neither an offset nor a period is specified, then one timer event occurs at program start, simultaneous with the **startup** event.

The period and offset are given by a number and a units, for example, `10 msec`. See the expressions documentation for allowable units. Consider the following example:

```
target Cpp;

main reactor Timer {
    timer t(0, 1s);

    reaction(t) {=
        std::cout << "Logical time is: " << get_logical_time() << std::endl
    =}
}
```

This specifies a timer named `t` that will first trigger at the start of execution and then repeatedly trigger at intervals of one second. Notice that the time units can be left off if the value is zero.

This target provides a built-in function for retrieving the logical time at which the reaction is invoked, FIXME . On most platforms (with the exception of some embedded platforms), the returned value is a 64-bit number representing the number of nanoseconds that have elapsed since January 1, 1970. Executing the above displays something like the following:

```
Logical time is 1648402121312985000.
Logical time is 1648402122312985000.
Logical time is 1648402123312985000.
...
```

The output lines appear at one second intervals unless the `fast` option has been specified.

Elapsed Time

The times above are a bit hard to read, so, for convenience, each target provides a built-in function to retrieve the *elapsed* time. For example:

```
target Cpp;

main reactor TimeElapsed {
    timer t(0, 1s);

    reaction(t) {=
        std::cout << "Elapsed logical time is " << get_elapsed_logical_time
    =}
}
```

See the Target Language Details for the full set of functions provided for accessing time values.

Executing this program will produce something like this:

```
Elapsed logical time is 0.
Elapsed logical time is 1000000000.
Elapsed logical time is 2000000000.
...
```

Comparing Logical and Physical Times

The following program compares logical and physical times:

```
target Cpp;

main reactor TimeLag {
    timer t(0, 1s);
    reaction(t) {=
        auto logical_time = get_elapsed_logical_time();
        auto physical_time = get_elapsed_physical_time();
        std::cout << "Elapsed logical time: " << logical_time
            << " physical time: " << physical_time
            << " lag: " << physical_time - logical_time <<  std::endl;
    =}
}
```

Execution will show something like this:

```
Elapsed logical time: 0, physical time: 855000, lag: 855000
Elapsed logical time: 1000000000, physical time: 1004714000, lag: 4714000
Elapsed logical time: 2000000000, physical time: 2004663000, lag: 4663000
Elapsed logical time: 3000000000, physical time: 3000210000, lag: 210000
...
```

In this case, the lag varies from a few hundred microseconds to a small number of milliseconds. The amount of lag will depend on the execution platform.

Simultaneity and Instantaneity

If two timers have the same *offset* and *period*, then their events are logically simultaneous. No observer will be able to see that one timer has triggered and the other has not.

A reaction is always invoked at a well-defined logical time, and logical time does not advance during its execution. Any output produced by the reaction will be **logically simultaneous** with the input. In other words, reactions are **logically instantaneous** (for an exception, see Logical Execution Time). Physical time, however, does elapse during execution of a reaction.

Timeout

By default, a Lingua Franca program will terminate when there are no more events to process. If there is a timer with a non-zero period, then there will always be more events to process, so the default execution will be unbounded. To specify a finite execution horizon, you can either specify a `timeout` target property or a `--timeout` command-line option. For example, the following `timeout` property will cause the above timer with a period of one second to terminate after 11 events:

```
target Cpp {
    timeout: 10 sec
}
```

Startup and Shutdown

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*.

To cause a reaction to be invoked at the end of execution, a special **shutdown** trigger is provided. Consider the following reactor, commonly used to build regression tests:

```
target Cpp;

reactor TestCount(start:int(0), stride:int(1), num_inputs:int(1)) {
    state count:int(start);
    state inputs_received:int(0);
    input x:int;

    reaction(x) {=
        auto value = *x.get();
        std::cout << "Received " <<  value << std::endl;
        if (value != count) {
            std::cerr << "ERROR: Expected: "<< count << std::endl;
            exit(1);
        }
        count += stride;
        inputs_received++;
    =}

    reaction(shutdown) {=
        std::cout << "Shutdown invoked." << std::endl;
        if (inputs_received != num_inputs) {
            std::cerr << "ERROR: Expected to receive " << num_inputs
                << " inputs, but got " << inputs_received << std::endl;
            exit(2);
        }
    =}
}
```

This reactor tests its inputs against expected values, which are expected to start with the value given by the `start` parameter and increase by `stride` with each successive input. It expects to receive a total of `num_inputs` input events. It checks the total number of inputs received in its **shutdown** reaction.

The **shutdown** trigger typically occurs at microstep 0, but may occur at a larger microstep. See Superdense Time and Termination.

Composing Reactors

Contained Reactors

Reactors can contain instances of other reactors defined in the same file or in an imported file. Assume the `Count` and `Scale` reactors defined in Parameters and State Variables are stored in files `Count.lf` and `Scale.lf`, respectively, and that the `TestCount` reactor from Time and Timers is stored in `TestCount.lf`. Then the following program composes one instance of each of the three:

```
target Cpp {
    timeout: 1 sec,
    fast: true
}

import Count from "Count.lf";
import Scale from "Scale.lf";
import TestCount from "TestCount.lf";

main reactor RegressionTest {
    c = new Count();
    s = new Scale(factor = 4);
    t = new TestCount(stride = 4, num_inputs = 11);
    c.y -> s.x;
    s.y -> t.x;
}
```

Diagrams

As soon as programs consist of more than one reactor, it becomes particularly useful to reference the diagrams that are automatically created and displayed by the Lingua Franca IDEs. The diagram for the above program is as follows:

In this diagram, the timer is represented by a clock-like icon, the reactions by chevron shapes, and the **shutdown** event by a diamond. If there were a **startup** event in this program, it would appear as a circle.

Creating Reactor Instances

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 is a comma-separated list of assignments:

```
<parameter_name> = <value>, ...
```

Like the default value for parameters, `<value>` can be a numeric constant, a string enclosed in quotation marks, a time value such as `10 msec`, target-language code enclosed in `{= ... =}`, or any of the list forms described in Expressions.

Connections

Connections between ports are specified with the syntax:

```
<source_port_reference> -> <destination_port_reference>
```

where the port references are either `<instance_name>.<port_name>` or just `<port_name>`, where the latter form is used for connections that cross hierarchical boundaries, as illustrated in the next section.

On the left and right of a connection statement, you can put a comma-separated list. For example, the above pair of connections can be written,

```
c.y, s.y -> s.x, t.x
```

The only constraint is that the total number of channels on the left match the total number on the right.

A destination port (on the right) can only be connected to a single source port (on the left). However, a source port may be connected to multiple destinations, as in the following example:

Lingua Franca provides a convenient shortcut for such multicast connections, where the above two lines can be replaced by one as follows:

```
(a.y)+ -> b1.x, b2.x
```

The enclosing `( ... )+` means to repeat the enclosed comma-separated list of sources however many times is needed to provide inputs to all the sinks on the right of the connection `->`.

Import Statement

An import statement has the form:

```
    import <classname> as <alias> from "<path>"
```

where `<classname>` and `<alias>` can be a comma-separated list to import multiple reactors from the same file. The `<path>` specifies another `.lf` file relative to the location of the current file. The `as <alias>` portion is optional and specifies alternative class names to use in the **new** statements.

Hierarchy

Reactors can be composed in arbitrarily deep hierarchies. For example, the following program combines the `Count` and `Scale` reactors within on `Container`:

```
target Cpp;
import Count from "Count.lf";
import Scale from "Scale.lf";
import TestCount from "TestCount.lf";

reactor Container(stride:int(2)) {
    output y:int;
    c = new Count();
    s = new Scale(factor = stride);
    c.y -> s.x;
    s.y -> y;
}

main reactor Hierarchy {
    c = new Container(stride = 4);
    t = new TestCount(stride = 4, num_inputs = 11);
    c.y -> t.x;
}
```

The `Container` has a parameter named `stride`, whose value is passed to the `factor` parameter of the `Scale` reactor. The line

```
s.y -> y;
```

establishes a connection across levels of the hierarchy. This propagates the output of a contained reactor to the output of the container. A similar notation may be used to propagate the input of a container to the input of a contained reactor,

```
x -> s.x;
```

Connections with Logical Delays

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

```
<source_port_reference> -> <destination_port_reference> after <time_val
```

where `<time_value>` can be any of the forms described in Expressions.

The **after** keyword specifies that the logical time of the event delivered to the destination port will be larger than the logical time of the reaction that wrote to source port. The time value is required to be non-negative, but it can be zero, in which case the input event at the receiving end will be one microstep later than the event that triggered it.

Physical Connections

A subtle and rarely used variant of the `->` connection is a **physical connection**, denoted `~>`. For example:

```
main reactor {
    a = new A();
    b = new B();
    a.y ~> b.x;
}
```

This is rendered in by the diagram synthesizer as follows:

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 (unless fast is set to `true`), 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).

Reactions and Methods

Reaction Order

A reactor may have multiple reactions, and more than one reaction may be enabled at any given tag. In Lingua Franca semantics, if two or more reactions of the same reactor are **simultaneously enabled**, then they will be invoked sequentially in the order in which they are declared. More strongly, the reactions of a reactor are **mutually exclusive** and are invoked in tag order primarily and declaration order secondarily. Consider the following example:

```
target Cpp {
    timeout: 3s
}
main reactor Alignment {
    state s:int(0);
    timer t1(100ms, 100ms);
    timer t2(200ms, 200ms);
    timer t4(400ms, 400ms);
    reaction(t1) {=
        s += 1;
    =}
    reaction(t2) {=
        s -= 2;
    =}
    reaction(t4) {=
        std::cout << "s = " << std::to_string(s) << std::endl;
    =}
}
```

Every 100 ms, this increments the state variable `s` by 1, every 200 ms, it decrements `s` by 2, and every 400 ms, it prints the value of `s`. When these reactions align, they are invoked in declaration order, and, as a result, the printed value of `s` is always 0.

Overwriting Outputs

Just as the reactions of the `Alignment` reactor overwrite the state variable `s`, logically simultaneous reactions can overwrite outputs. Consider the following example:

```
target Cpp;
reactor Overwriting {
    output y:int;
    state s:int(0);

    timer t1(100ms, 100ms);
    timer t2(200ms, 200ms);

    reaction(t1) -> y {=
        s += 1;
        y.set(s);
    =}
    reaction(t2) -> y {=
        s -= 2;
        y.set(s);
    =}
}
```

Here, the reaction to `t1` will set the output to 1 or 2, but every time it sets it to 2, the second reaction (to `t2`) will overwrite the output with the value 0. As a consequence, the outputs will be 1, 0, 1, 0, ... deterministically.

Reacting to Outputs of Contained Reactors

A reaction may be triggered by the an input to the reactor, but also by an output of a contained reactor, as illustrated in the following example:

```
target Cpp;
import Overwriting from "Overwriting.lf";

main reactor {
    s = new Overwriting();
    reaction(s.y) {=
        auto is_correct = [](auto value){
            return value == 0 || value == 1;
        };

        if (s.y.is_present() && !is_correct(*s.y.get())) {
            std::cout << "Output shoudl only be 0 or 1!" << std::endl;
        }
    =}
}
```

This instantiates the above `Overwriting` reactor and monitors its outputs.

Triggering Contained Reactors

A reaction can set the input of a contained reactor, thereby triggering its reactions, as illustrated in the following example:

```
target Cpp
reactor Inside {
  input x: int
  reaction(x) {=
    std::cout << "Received " << std::to_string(*x.get()) << std::endl;
  =}
}
main reactor {
  i = new Inside()
  reaction(startup) -> i.x {=
    i.x.set(42);
  =}
}
```

The reaction to **startup** declares the input port of the inside reactor as an effect and then sets it with value 42. This will cause the inside reactor's reaction to execute and print `Received 42`.

Method Declaration

Causality Loops

Cycles

The interconnection pattern for a collection of reactors can form a cycle, but some care is required. Consider the following example:

```
target Cpp;
reactor A {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        // ... something here ...
    =}
}
reactor B {
    input x:int;
    output y:int;
    reaction(x) {=
        // ... something here ...
    =}
    reaction(startup) -> y {=
        // ... something here ...
    =}
}
main reactor {
    a = new A();
    b = new B();
    a.y -> b.x;
    b.y -> a.x;
}
```

This program yields the following diagram:

The diagram highlights a **causality loop** in the program. At each tag, in reactor `B`, the first reaction has to execute before the second if it is enabled, a precedence indicated with the red dashed arrow. But the first can't execute until the reaction of `A` has executed, and that reaction cannot execute until the second reaction `B` has executed. There is no way to satisfy these requirements, so the tools refuse to generated code.

Cycles with Delays

One way to break the causality loop and get an executable program is to introduce a logical delay into the loop, as shown below:

```
target Cpp;
reactor A {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        // ... something here ...
    =}
}
reactor B {
    input x:int;
    output y:int;
    reaction(x) {=
        // ... something here ...
    =}
    reaction(startup) -> y {=
        // ... something here ...
    =}
}
main reactor {
    a = new A();
    b = new B();
    a.y -> b.x after 0;
    b.y -> a.x;
}
```

Here, we have used a delay of 0, which results in a delay of one microstep. We could equally well have specified a positive time value.

Reaction Order

Frequently, a program will have such cycles, but you don't want a logical delay in the loop. To get a cycle without logical delays, the reactions need to be reordered, as shown below:

```
target Cpp;
reactor A {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        // ... something here ...
    =}
}
reactor B {
    input x:int;
    output y:int;
    reaction(startup) -> y {=
        // ... something here ...
    =}
    reaction(x) {=
        // ... something here ...
    =}
}
main reactor {
    a = new A();
    b = new B();
    a.y -> b.x;
    b.y -> a.x;
}
```

There is no longer any causality loop.

Extending Reactors

Extending a Base Reactor

The Cpp target does not yet support extending reactors.

Actions

Action Declaration

An action declaration has one of the following forms:

```
logical action <name>(<min_delay>, <min_spacing>, <policy>)
physical action <name>(<min_delay>, <min_spacing>, <policy>)
```

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`. The `min_delay` and `min_spacing` are time values. The `policy` argument is a string that can be one of the following: `"defer"` (the default), `"drop"`, or `"replace"`. Note that the quotation marks are needed.

Logical Actions

Timers are useful to trigger reactions once or periodically. Actions are used to trigger reactions more irregularly. An action, like an output or input port, can carry data, but unlike a port, an action is visible only within the reactor that defines it.

There are two kinds of actions, **logical** and **physical**. A **logical action** is used by a reactor to schedule a trigger at a fixed logical time interval *d* into the future. The time interval *d*, which is called a **delay**, is relative to the logical time *t* at which the scheduling occurs. If a reaction executes at logical time *t* and schedules an action `a` with delay *d*, then any reaction that is triggered by `a` will be invoked at logical time *t* + *d*. For example, the following reaction schedules something (printing the current elapsed logical time) 200 msec after an input `x` arrives:

```
target Cpp;

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

Here, the delay is specified in the call to schedule within the target language code. Notice that in the diagram, a logical action is shown as a triangle with an **L**. Logical actions are always scheduled within a reaction of the reactor that declares the action.

The time argument is required to be non-negative. If it is zero, then the action will be scheduled one **microstep** later. See Superdense Time.

An action may have a data type, in which case, a variant of the `schedule()` function can be used to specify a **payload**, a data value that is carried from where the `schedule()` function is called to the reaction that is triggered by the action. See the Target Language Details.

Physical Actions

A **physical action** is used to schedule reactions at logical times determined by the local physical clock. If a physical action with delay *d* is scheduled at *physical* time *T*, then the *logical time* assigned to the event is *T* + *d*. For example, the following reactor schedules the physical action `p` to trigger at a **logical time** equal to the **physical time** at which the input `x` arrives:

```
target Cpp;

reactor Physical {
    input x:int;
    physical action a;

    reaction(x) -> a {=
        a.schedule(0ms);
    =}

    reaction(a) {=
        auto elapsed_time = get_elapsed_logical_time();
        std::cout << "Action triggered at logical time " << elapsed_time <<
    =}
}
```

If you drive this with a timer, using for example the following structure:

then running the program will yield an output something like this:

```
Action triggered at logical time 201491000 nsec after start.
Action triggered at logical time 403685000 nsec after start.
Action triggered at logical time 603669000 nsec after start.
...
```

Here, logical time is lagging physical time by a few milliseconds. Note that, unless the fast option is given, logical time *t* chases physical time *T*, so $t < T$. Hence, the event being scheduled in the reaction to input `x` is assured of being in the future in logical time.

Whereas logical actions are required to be scheduled within a reaction of the reactor that declares the action, physical actions can be scheduled by code that is outside the Lingua Franca system. For example, some other thread or a callback function may call `schedule()`, passing it a physical action. For example:

```
target Cpp;
main reactor {
    private preamble {=
        #include <thread>
    =}

    state thread: std::thread;
    physical action a:int;

    reaction(startup) -> a {=
        // Start a thread to schedule physical actions.
        thread = std::thread(&
        });
    =}

    reaction(a) {=
        auto elapsed_time = get_physical_time();
        std::cout << "Action triggered at logical time" << elapsed_time <<"
    =}
}
```

Physical actions are the mechanism for obtaining input from the outside world. Because they are assigned a logical time derived from the physical clock, their logical time can be interpreted as a measure of the time at which some external event occurred.

Triggering Time for Actions

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 in the action declaration, and whether the action is physical or logical.

For a **logical** action `a`, the tag assigned to the event resulting from a call to `schedule()` is computed as follows. First, let *t* be the *current logical time*. For a logical action, *t* is just the logical time at which the reaction calling `schedule()` is called. The **preliminary time** of the action is then just *t* + `<min_delay>` + `<offset>`. This preliminary time may be further modified, as explained below.

For a **physical** action, the preliminary time is similar, except that *t* is replaced by the current *physical* time *T* when `schedule()` is called.

If a `<min_spacing>` has been declared, then it gives a minimum logical time interval between the tags of two subsequently scheduled events. If the preliminary time is closer than `<min_spacing>` to the time of the previously scheduled event (if there is one), then `<policy>` (if supported by the target) determines how the minimum spacing constraint is enforced.

Testing an Action for Presence

When a reaction is triggered by more than one action or by an action and an input, it may be necessary to test within the reaction whether the action is present. Just like for inputs, this can be done in the C++ target with `a.is_present()`, where `a` is the name of the action.

Superdense Time

Tag vs. Time

The model of time in Lingua Franca is a bit more sophisticated than we have hinted at. Specifically, a **superdense** model of time is used. In particular, instead of a **timestamp**, LF uses a **tag**, which consists of a **logical time** *t* and a **microstep** *m*.

A **logical action** may have a `<min_delay>` of zero, and the `<offset>` argument to the `schedule()` function may be zero. In this case, the call to `schedule()` appears to be requesting that the action trigger at the *current logical time*. Here is where superdense time comes in. The action will indeed trigger at the current logical time, but one microstep later. Consider the following example:

```
target Cpp;
main reactor {
    state count:int(1);
    logical action a;
    reaction(startup, a) -> a {=
        std::cout << count << " Logical time is " << get_logical_time() <<
        if (count++ < 5) {
            a.schedule();
        }
    =}
}
```

Executing this program will yield something like this:

```
1. Logical time is 1649607749415269000. Microstep is 0.
2. Logical time is 1649607749415269000. Microstep is 1.
3. Logical time is 1649607749415269000. Microstep is 2.
4. Logical time is 1649607749415269000. Microstep is 3.
5. Logical time is 1649607749415269000. Microstep is 4.
```

Notice that the logical time is not advancing, but the microstep is (the logical time, in this case, gives the number of nanoseconds that have elapsed since January 1, 1970). The general rule is that **every** call to `schedule()` advances the tag by at least one microstep.

Logical Simultaneity

Two events are **logically simultaneous** only if *both* the logical time and the microstep are equal. The following example illustrates this:

```
target Cpp;

reactor Destination {
    input x:int;
    input y:int;
    reaction(x, y) {=
        std::cout << "Time since start: " << get_elapsed_logical_time() <<
        if (x.is_present()) {
            std::cout << "x is present" << std::endl;
        }
        if (y.is_present()) {
            std::cout << "y is present" << std::endl;
        }
    =}
}

main reactor {
    logical action repeat;
    d = new Destination();
    reaction(startup) -> d.x, repeat {=
        d.x.set(1);
        repeat.schedule(0ms);
    =}
    reaction(repeat) -> d.y {=
        d.y.set(1);
    =}
}
```

The `Destination` reactor has two inputs, `x` and `y`, and it reports in a reaction to either input what is the logical time, the microstep, and which input is present. The main reactor reacts to **startup** by sending data to the `x` input of `Destination`. It then schedules a `repeat` action with an `<offset>` of zero. The `repeat` reaction is invoked **strictly later**, one **microstep** later. The output printed, therefore, will look like this:

```
Time since start: 0, microstep: 0
  x is present.
Time since start: 0, microstep: 1
  y is present.
```

The reported 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.

Alignment of Logical and Physical Times

Recall that in Lingua Franca, logical time "chases" physical time, invoking reactions at a physical time close to their logical time. For that purpose, the microstep is ignored.

Modal Reactors

Deadlines

Lingua Franca includes a notion of a **deadline**, which is a constraint on the 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* time 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.

Purposes for Deadlines

A deadline in an LF program serves two purposes. First, it can guide scheduling in that a scheduler may prioritize reactions with deadlines over those without or those with longer deadlines. For this purpose, if a reaction has a deadline, then all upstream reactions on which it depends (without logical delay) inherit its deadline. Hence, those upstream reactions will also be given higher priority.

Second, the deadline mechanism provides a **fault handler**, a section of code to invoke when the deadline requirement is violated. Because invocation of the fault handler depends on factors beyond the control of the LF program, an LF program with deadlines becomes **nondeterministic**. The behavior of the program depends on the exact timing of the execution.

There remains the question of when the fault handler should be invoked. By default, deadlines in LF are **lazy**, meaning that the fault handler is invoked at the logical time of the event triggering the reaction whose deadline is missed. Specifically, the possible violation of a deadline is not checked until the reaction with the deadline is ready to execute. Only then is the determination made whether to invoke the regular reaction or the fault handler.

An alternative is an **eager deadline**, where a fault handler is invoked as soon as possible after a deadline violation becomes inevitable. With an eager deadline, if an event with tag (t, m) triggers a reaction with deadline D, then as soon as the runtime system detects that physical time $T > t + D$, the fault handler becomes enabled. This can occur at a logical time *earlier* than t. Hence, a fault handler may be invoked at a logical time earlier than that of the event that triggered the fault.

Note: As of this writing, eager deadlines are not implemented in any LF target language, so all deadlines are lazy.

Lazy Deadline

A lazy deadline is specified as follows:

```
target Cpp;

reactor Deadline {
    input x:int;
    output d:int; // Produced if the deadline is violated.
    reaction(x) -> d {=
        std::cout << "Normal reaction." << std::endl;
    =} deadline(10ms) {=
        std::cout << "Deadline violation detected." << std::endl;
        d.set(*x.get());
    =}
}
```

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. This reactor can be tested as follows:

```
target Cpp;
import Deadline from "Deadline.lf";

main reactor {
    logical action a;
    d = new Deadline();
    reaction(startup) -> d.x, a {=
        d.x.set(0);
        a.schedule(0ms);
    =}

    reaction(a) -> d.x {=
        d.x.set(0);
        std::this_thread::sleep_for(20ms);
    =}

    reaction(d.d) {=
        std::cout << "Deadline reactor produced an output." << std::endl;
    =}
}
```

Running this program will result in the following output:

```
Normal reaction.
Deadline violation detected.
Deadline reactor produced an output.
```

The first reaction of the `Deadline` reactor does not violate the deadline, but the second does. Notice that the sleep in the **main** reactor occurs *after* setting the output, but because of the deterministic semantics of LF, this does not matter. The actual value of an output cannot be known until every reaction that sets that output *completes* its execution. Since this reaction takes at least 20 msec to complete, the deadline is assured of being violated.

Notice that the deadline is annotated in the diagram with a small clock symbol.

Deadline Violations During Execution

Whether a deadline violation occurs is checked only *before* invoking the reaction with a deadline. What if the reaction itself runs for long enough that the deadline gets violated *during* the reaction

execution? For this purpose, a target-language function is provided to check whether a deadline is violated during execution of a reaction with a deadline.

Consider this example:

```
WARNING: No source file found: ../code/cpp/src/CheckDeadline.lf
```

Multiports and Banks

Lingua Franca provides a compact syntax for ports that can send or receive over multiple channels and another syntax for multiple instances of a reactor class. These are respectively called **multiports** and **banks of reactors**.

Multiports

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

where `<width>` is a positive integer. This can be given either as an integer literal or a parameter name. The width can also be given by target code enclosed in `{=...=}`. Consider the following example:

```
target Cpp;
reactor Source {
    output[4] out:int;
    reaction(startup) -> out {=
        for(auto i = 0ul; i < out.size(); i++) {
            out[i].set(i);
        }
    =}
}
reactor Destination {
    input[4] in:int;
    reaction(in) {=
        int sum = 0;
        for (auto i = 0ul; i < in.size(); i++) {
            if (in[i].is_present()){
                sum += *in[i].get();
            }
        }
        std::cout << "Sum of received: " << sum << std::endl;
    =}
}
main reactor {
    a = new Source();
    b = new Destination();
    a.out -> b.in;
}
```

Executing this program will yield:

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

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. Notice that the generated diagram shows multiports with hollow triangles. Whether it shows the widths is controlled by an option in the diagram generator.

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.

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.

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 {=
        ...
    =}
}
```

Parameters to the main reactor can be overwritten on the command line interface when running the generated program. As a consequence, the scale of the application can be determined at run time rather than at compile time.

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:

```
main reactor {
    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 input 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:

```
main reactor {
    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 {
    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**. If such a parameter is defined for the reactor, then when the reactor is instantiated 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:

```
target Cpp;

reactor MultiportSource(
    bank_index:int(0)
) {
    timer t(0, 200 ms);
    output out:int;
    state s:int(0);

    reaction(t) -> out {=
        out.set(s);
        s += bank_index;
    =}
}
```

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

```
main reactor(
    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 Cpp;
reactor Child (
    bank_index:int(0)
) {
    reaction(startup) {=
        std::cout << "My bank index:" << bank_index << std::endl;
    =}
}
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: 0.
My bank index: 1.
My bank index: 0.
My bank index: 1.
```

The order of these outputs will be nondeterministic if the execution is multithreaded (which it will be by default) because there is no dependence between the reactions, and, hence, they can execute in parallel.

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

```
target Cpp;
reactor Child (
    bank_index:int(0),
    parent_bank_index:int(0)
) {
    reaction(startup) {=
        std::cout <<"My bank index: " << bank_index << " My parent's bank i
    =}
}
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 bank index: 1. My parent's bank index: 1.
My bank index: 0. My parent's bank index: 0.
My bank index: 0. My parent's bank index: 1.
My bank index: 1. My parent's bank index: 0.
```

Again, note that the order of these outputs is nondeterministic.

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 Cpp;
reactor Child (
    bank_index:int(0),
    parent_bank_index:int(0)
) {
    output out:int;
    reaction(startup) -> out {=
        out.set(parent_bank_index * 2 + bank_index);
    =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
    reaction(c.out) {=
        for (auto i = 0ul; i < c.size(); i++) {
            std::cout << "Received " << *c[i].out.get() <<" from child " <<
        }
    =}
}
main reactor {
    p = new[2] Parent();
}
```

Running this program will give something like the following:

```
Received 0 from child 0.
Received 1 from child 1.
Received 2 from child 0.
Received 3 from child 1.
```

Note that `c.size()` can be used to get the width of the bank `c`.

Combining Banks and Multiports

Banks of reactors may be combined with multiports, as in the following example:

```
target Cpp;

reactor Source {
    output[3] out: int;
    reaction(startup) -> out {=
        for(int i = 0; i < out.size(); i++) {
            out[i].set(i);
        }
    =}
}
reactor Destination(
    bank_index:int(0)
) {
    input in:int;
    reaction(in) {=
        std::cout << "Destination " << bank_index << " received " << *in.ge
    =}
}

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 result of the program will be something like the following:

```
Destination 0 received 0.
Destination 1 received 1.
Destination 2 received 2.
```

Again, the order is nondeterministic in a multithreaded context.

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 following 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 {=
                ... write to out ...
        =}
}
reactor Destination {
        input in:int;
        reaction(in) {=
                ... read from 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.

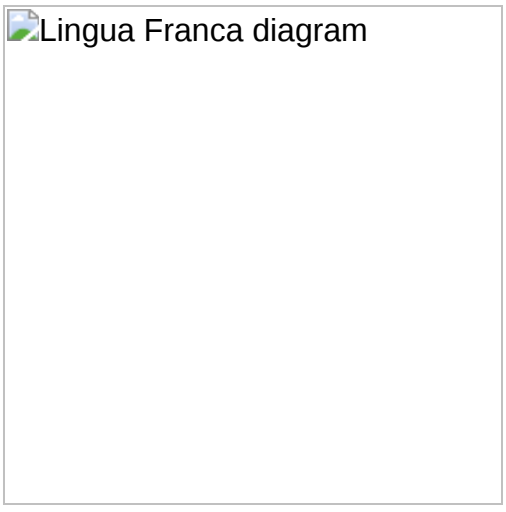
Interleaved Connections

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 *n*-th reactor instance within the destination 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 following example:

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

    reaction (startup) -> out {=
        out[1].set(42);
        std::cout << "Bank index " << bank_index << " sent 42 on channel 1.
    =}

    reaction (in) {=
        for (auto i = 0ul; i < in.size(); i++) {
            if (in[i].is_present()) {
                std::cout << "Bank index " << bank_index
                    << " received " << *in[i].get() << " on channel" << std
            }
        }
    =}
}
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. Running this program gives something like the following:

```
Bank index 0 sent 42 on channel 1.
Bank index 1 sent 42 on channel 1.
Bank index 2 sent 42 on channel 1.
Bank index 3 sent 42 on channel 1.
Bank index 1 received 42 on channel 0.
Bank index 1 received 42 on channel 1.
Bank index 1 received 42 on channel 2.
Bank index 1 received 42 on channel 3.
```

In bank index 1, the 0-th channel receives 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 from which the input comes.

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, for each bank index, iterating over 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 how this pattern would look without banks or multiports:

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

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

Generic Reactors

Generic Reactors

Sometimes it is useful to implement a generic pattern without knowing the concrete types used. For instance, it could be useful to implement a delay reactor that forwards all values it receives with a fixed delay. For this pattern, it is not required to know the concrete type in advance and we would like to reuse the same logic for different types. This can be achieved with generic reactors in LF. Consider the following example:

```
reactor Delay<T>(delay: time(0)) {
    input in: T
    output out: T
    logical action a(delay): T

    reaction(a) -> out {= out.set(a.get()); =}

    reaction(in) -> a {= a.schedule(in.get(), delay); =}
}

main reactor {
    d = new Delay<int>(delay = 100 msec)

    reaction(startup) -> d.in {= d.in.set(42); =}
    reaction(d.out) -> d.out {= std::cout << "received " << *d.in.get(); =}
}
```

The example above defines a generic reactor `Delay` which receives a type parameter `T`. Its input, output and logical action are all of type `T`. The logic implemented in the reactions is straight forward. The reaction to `in` schedules the logical action `a` with the configured delay and the received value. The reaction to `a` simply forwards this value to the output port at a later tag. The concrete type `T`, however, is not relevant for this implementation and will be filled in only, when the reactor is instantiated. In our example, the main reactor instantiates `Delay`, specifying `int` as the type to be assigned to `T`. In consequence, we can set an integer on `d`'s input port and receive an integer on its output. If we wanted instead to delay a string, we can do this as follows:

```
main reactor {
    d = new Delay<{=std::string=}>(delay = 100 msec)

    reaction(startup) -> d.in {= d.in.set("foo"); =}
    reaction(d.out) -> d.out {= std::cout << "received " << *d.in.get(); =}
}
```

Reactor definitions may also specify multiple type parameters. Moreover, type parameters are not limited to ports and actions, but can also be used in state variables, parameters, or methods. For instance, we can define the following reactor:

```
reactor Generic<T, U, V>(bar: T) {
  state baz: U

  input in: V

  method (x: T, y: U): V {= /* ... */ =}
}
```

This reactor could be instantiated for example like this:

```
g = new Generic<float, int, bool>
```

Preambles

Preamble

Reactions may contain arbitrary target-language 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 `charconv` header from the c++ standard library to convert a string to an integer:

```
target Cpp;

main reactor {
    private preamble {=
        #include <charconv>
        #include <string>
    =}

    timer t;
    reaction(t) {=
        std::string raw = "42";
        std::size_t number;

        auto result = std::from_chars(raw.data(), raw.data() + raw.size(),
        if (result.ec == std::errc::invalid_argument) {
            std::cerr << "Could not convert.";
        } else {
            std::cout << "Converted string: " << raw << " to integer: " <<
        }
    =}
}
```

This will print:

```
[INFO]  Starting the execution
Converted string: 42 to integer: 42
[INFO]  Terminating the execution
```

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.

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

    private preamble {=
        auto add_42(int i) noexcept -> int {
            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.

Distributed Execution

Termination

Shutdown Reactions

There are several mechanisms for terminating a Lingua Franca in an orderly fashion. All of these mechanisms result in a **final tag** at which any reaction that declares **shutdown** as a trigger will be invoked (recall that a **tag** is a tuple (**logical time**, **microstep**)). Other reactions may also be invoked at this final tag, and the order in which reactions are invoked will be constrained by the normal precedence rules.

If a reaction triggered by **shutdown** produces outputs, then downstream reactors will also be invoked at the final tag. If the reaction schedules any actions by calling `schedule()`, those will be ignored. In fact, any event after the final tag will be ignored. After the completion of the final tag, the program will exit.

There are four ways to terminate a program:

- **Timeout**: The program specifies the last logical time at which reactions should be triggered.
- **Starvation**: At the conclusion of some tag, there are no events in the event queue at future tags.
- **Stop request**: Some reaction requests that the program terminate.
- **External signal**: Program is terminated externally using operating services like control-C or `kill`.

We address each of these in turn.

Timeout

The target property `timeout` specifies the last logical time at which reactions should be triggered. The last invocation of reactions will be at tag (`timeout`, 0).

There is a significant subtlety when using physical connections, which are connections using the syntax `~>`. Such connections specify that the tag at the receiving end will be based on the physical time at which the message is received. If the tag assigned at the receiving end is greater than the final tag, then the message is lost. Hence, **messages sent near the `timeout` time are likely to be lost!**

Starvation

If a Lingua Franca program has no physical actions, and if at any time during execution there are no future events waiting to be processed, then there is no possibility for any more reactions to occur and the program will exit. This situation is called **starvation**. If there is a **timer** anywhere in the program with a period, then this condition never occurs.

One subtlety is that reactions triggered by **shutdown** will be invoked one microstep later than the last tag at which there was an event. They cannot be invoked at the same tag because it is only after that last tag has completed that the runtime system can be sure that there are no future events. It would not be correct to trigger the **shutdown** reactions at that point because it would be impossible to respect the required reaction ordering.

Stop Request

If a reaction calls the built-in `request_stop()` function, then it is requesting that the program cease execution as soon as possible. This cessation will normally occur in the next microstep. The current tag will be completed as normal. Then the tag will be advanced by one microstep, and reactions triggered by **shutdown** will be executed, along with any other reactions with triggers at that tag, with all reactions executed in precedence order.

External Signal

A control-C or other kill signal to a running Lingua Franca program will cause execution to stop immediately.