Work-In-Progress: Real-Time Reactors in C

Marten Lohstroh and Edward A. Lee
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA, USA
{marten, eal}@berkeley.edu

***Abstract*—This paper describes an implementation in progress of a C-based framework for execution of deterministic, concurrent, real-time software components called "reactors." The component interfaces and their interconnections are given in a coordination language called Lingua Franca, while the work done by the components is given in ordinary C. The implementation described here can exploit multiple cores and is capable of realizing rate monotonic and earliest deadline first scheduling policies.**

***Index Terms*—Concurrency control, Distributed computing, Real-time systems**

I. Introduction

Lingua Franca (LF) is a recently introduced coordination language [1] for the definition and composition of reactors [2]. Reactors are concurrent real-time software components that are triggered either periodically or in reaction to events. The model of computation combines timestamps with synchronous-reactive principles to emphasize predictable and analyzable behavior. LF is a polyglot framework, intended to be used with a variety of programming languages. This paper describes a work-in-progress, a preliminary implementation of LF used with C.[1] Because C is a rather low-level language, lacking a strong type system, memory management, and support for object-oriented design, it presents a number of challenges. On the other hand, C is the most universally supported language for embedded system design, and it runs efficiently on processors ranging from the smallest 8-bit microcontrollers to sophisticated 64-bit multicore processors. A major goal of this experiment is to quantify the minimal cost of supporting the deterministic concurrency model of Lingua Franca, a goal for which C is a suitable choice.

Two implementations are described in this paper. The first is suitable for very low-level embedded controllers, even those lacking an operating system. It relies on a small set of standard C libraries (`stdio.h`, `stdlib.h`, `string.h`, `time.h`, and `errno.h`). Any embedded platform with a C compiler and an implementation of these libraries can run the code generated by the LF compiler. This implementation is suitable for embedded applications where most activities are periodic.

The second implementation adds one more library requirement (`pthread.h`). The addition of this library enables multi-core execution and integration of asynchronous external events (e.g., those generated by an interrupt request). A third implementation is planned, but not yet implemented. It adds a requirement for clocks that are synchronized across a network and extends deterministic parallel execution to deterministic distributed execution based on the principles of PTIDES [3].

As a coordination language, LF governs the interactions and concurrent execution of chunks of C code. For the purposes of this experiment, we make no attempt to limit what those chunks C code can do, and instead assume that they conform to the principles of reactors. For example, each reactor instance has private state, and reactors cannot access each other's state. In C, this is difficult to enforce, and enforcement would likely add overhead. Hence, we simply assume that the chunks of C code are well behaved. This is consistent with identifying a minimum overhead run time implementation. Better safety properties could be achieved by either code generating the chunks of C code from a safer language or using LF with a different target language, such as Java or Rust. But that is out of scope for this paper.

Our compiler is built in Eclipse with Xtext. This combination provides a syntax-directed editor that runs within Eclipse and one that runs in a browser, plus a code generator that runs within Eclipse or from the command line. The code generator takes as input a Lingua Franca program and produces a standalone C program that includes all the required libraries.

The C runtime consists of about 1,000 lines of extensively commented code and occupies tens of kilobytes for a minimal application, making it suitable for deeply embedded platforms. We have tested it on Linux, Windows, and Mac platforms, as well as a couple of bare-iron platforms. On platforms that support pthreads (POSIX threads), it transparently exploits multiple cores while preserving determinism. The framework includes features for real-time execution and is particularly well suited to take advantage of platforms with predictable execution times, such as PRET machines [4].

II. The C Target

A Lingua Franca component is called a **reactor**. Reactors can have input ports, actions, and timers, all of which can trigger reactions. They can also have output ports, local state, parameters, and an ordered list of reactions.

A reactor may contain other reactors and manage their connections. The connections define the flow of events, and two reactors can be connected only if they are contained by the same reactor. An output port may be connected to multiple

The work in this paper was supported in part by the National Science Foundation (NSF), award #CNS-1836601 (Reconciling Safety with the Internet) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Camozzi Industries, Denso, Ford, Siemens, and Toyota.

[1]More detailed documentation, the source code, and test programs can be found at https://github.com/icyphy/lingua-franca/.

input ports, but an input port can only be connected to one output port.

Reactions are triggered by **input** events that a reactor receives from another reactor, by **timer** events, or by **action** events (events that are either generated internally by a reactor to trigger a future computation or generated by some external event such as an interrupt). Each of these triggers (input, timer, and action events) has a timestamp, a value on a logical timeline. Reactions are triggered in timestamp order. Each input port, timer, and action can have at most one such event at any logical time. And each such event may carry a value that is accessible by the triggered reactions.

A **reaction** is a procedure in a target language (C, in this case) 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 input events that it observes and output events 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).

Successive invocations of any single reaction occur at strictly increasing logical times. Any events that are not read by a reaction triggered at the timestamp of the event are lost (there is no queueing of events). Of values consecutively written to the same port at the same timestamp, only the last is witnessed by other reactors.

The execution of any two reactions of a reactor are mutually exclusive (atomic with respect to one another). Moreover, any two reactions of the same reactor 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. Reactions belonging to distinct reactors that are not dependent on one another can be executed in parallel on a multicore machine.

An LF program is deterministic unless the reactions written in the target language explicitly introduce nondeterminism, for instance, by reporting readings from some I/O device. Given the same input data, a composition of reactors has exactly one correct behavior. This makes LF programs more testable.

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

```
1 target C;
2 main reactor HelloWorld {
3     timer t;
4     reaction(t) {=
5         printf("Hello World.\n");
6     =}
7 }
```

The first line specifies the target language (C). The second line defines a "main" reactor instance that will execute. `HelloWorld` has a timer named `t` with no parameters, meaning it triggers exactly once at the start of execution. To specify a periodic trigger, this line could be replaced with, for example: `timer t(50 usec, 100 msec)`. This specifies that the timer will trigger first 50 microseconds after the start of execution and then trigger periodically with a period equal to 100 milliseconds. Note that these times are *logical times* that will be aligned on a best-effort basis with physical time, and the accuracy of such alignment will depend on the real-time capabilities of the execution platform. But since these are logical times, there is a key property that does not depend on the execution platform. If two timers that generate events at the same logical time, then those events are *logically simultaneous*, meaning that any observer (i.e., downstream reactor) will see that either both events are present or neither is. In addition, if two timers generate events at different logical times, then reactions to these events are guaranteed to be ordered according to the timestamps.[2]

Lines 4-6 specify a reaction that is triggered by `t`. The body of the reaction is C code that is not parsed or analyzed by the LF compiler, delimited by $\{= \cdots =\}$.

Reactors are composed hierarchically by connecting outputs to inputs using a pattern like this:

```
1 reactor A {
2   output y;
3   ...
4 }
5 reactor B {
6   input x;
7   ...
8 }
9 main reactor C {
10   a = new A();
11   b = new B();
12   a.y -> b.x;
13 }
```

Here, reactor `C` contains two reactor instances, one of class `A` and the other of class `B`. The output of the instance of `A` is connected to the input of the instance of class `B`. This ensures that at any logical time where a reaction of `A` produces an output, that reaction will execute to completion before the reaction in `B` that is triggered by this input is invoked. This precedence constraint is enforced even across processors.

A. Simultaneity

The following illustrates the principle of simultaneity:

```
1 reactor Add {
2     input in1:int;
3     input in2:int;
4     output out:int;
5     reaction(in1, in2) -> out {=
6         int result = 0;
7         if (in1_is_present) result += in1;
8         if (in2_is_present) result += in2;
9         set(out, result);
10    =}
11 }
```

A reaction executes at a logical time, and at that logical time, inputs to the reactor are either present or absent. In the above, the reaction is triggered at a logical time when either input `in1` or input `in2` is present (or both). Line 5 specifies these triggering conditions and also specifies that this reaction may yield an event via the output port named 'out'. For the C target, the boolean variables `in1_is_present` and `in2_is_present` are automatically provided by the code generator for testing for the presence of inputs. Lingua Franca guarantees that when two events with the same timestamp are routed to these two inputs, then regardless of the origin of those events, even if they come from another processor, the reaction will see both present.

[2]This strict ordering constraint can be relaxed when reactions have no mutual dependencies, enabling greater parallelism.

B. Time

In our run time implementation, a timestamp is a 64-bit integer specifying a 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 2262. 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.)

Working with nanoseconds can be tedious if you are interested in longer durations. For convenience, LF provides a **time** datatype that accepts units, such as the `100 msec` below. Since this syntax will not work with C, 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)`.

C. Scheduling Delayed Reactions

The C runtime provides a function `schedule(action, extraDelay, value)` that can be called from the target code. Consider the following reactor:

```
1  target C;
2  main reactor Clock {
3      timer t(0, 1 sec);
4      logical action a(100 msec);
5      reaction(t) -> a {=
6          schedule(a, 0, NULL);
7      =}
8      reaction(a) {=
9          printf("Nanoseconds since start: %lld.\n",
               get_elapsed_logical_time());
10     =}
11 }
```

This reactor defines a **logical action**, a trigger with a timestamp that is larger than the logical time at which the trigger is scheduled by a specified amount. This will produce:

```
Nanoseconds since start: 100000000.
Nanoseconds since start: 1100000000.
Nanoseconds since start: 2100000000.
...
```

With a period of one second, this reactor schedules the occurrence of an action event 100 msec later. The action `a` will be triggered at a logical time 100 msec after the timer triggers. This will trigger the second reaction, which will use the `get_elapsed_logical_time` function to determine how much logical time has elapsed since the start of execution.

The times reported by the program output are logical times, and hence are highly regular. It is is also possible to define a **physical action**, in which case the timestamp of the action will be derived from a local physical clock. This is useful for assigning meaningful timestamps to externally triggered events, such as those caused by an interrupt. Because chains of reactions triggered by such events are logically instantaneous, enforcing end-to-end real-time requirements between reactions is a matter of again comparing the current logical time to the current physical time once the last reaction in the chain is ready to execute.

Physical actions (and any reactions that directly or indirectly depend on them) are not allowed to trigger any reactions before the physical clock matches the timestamp of the action. Reactions to a logical action, however, do not necessarily need to wait for physical time to match the time stamp. Provided the side effects of such reactions are not observable outside of the reactor model (e.g., through a terminal or a physical actuator), they can execute "ahead of time," possibly enhancing performance, increasing parallelism, and making it easier to meet deadlines (see below).

D. 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 the reactions to some event must occur within some physical-time interval measured from the logical time of the event. If an event has timestamp 12 noon on August 8, 2019 and a reaction triggered by it has a deadline of one hour, then the reaction to the event is required to be invoked before the physical-time clock of the execution platform reaches 1 PM on August 8, 2019. If a deadline is violated, then instead of allowing the tardy event to trigger the reaction, the code in the body of the attached deadline handler is executed. For example:

```
1  reactor Sensor {
2      physical action a:int;
3      output y:int;
4      // ...
5      reaction(a) -> y {=
6          set(y, a->value);
7      =}
8  }
9  reactor Actuator {
10     input x:int;
11     reaction(x) {=
12         // Time-sensitive code
13     =} deadline(100 msec) {=
14         printf("**** Deadline miss detected.\n");
15     =}
16 }
17 main reactor Composite {
18     s = new Sensor();
19     d = new Actuator();
20     s.y -> d.x;
21 }
```

The above program assigns a deadline of 100 msec to a reaction of reactor `d`, an instance of reactor class `Actuator`. If `d`'s reaction to that input is not invoked within 100 msec of the timestamp of the input, then rather than executing the time-sensitive code in the reaction, the deadline violation is handled (in this case it just prints a message).

The presence of such deadlines in the LF code enables the code generator to synthesize earliest-deadline-first scheduling policies. The fact that dependencies between reactions are also known to the code generator enables inheritance of the resulting priorities by all upstream reactions that may directly or indirectly trigger the reaction with a deadline.

III. Execution

Our implementation uses two queues, an **event queue**, which is sorted by timestamp, and a **reaction queue**, which is sorted by priority. The event queue contains triggers (timers

and actions) that are to occur at a future logical time. The reaction queue contains reactions to be executed at a fixed logical time. In ensures that reactions that depend on one another are invoked in the proper order. Specifically, during code generation, an acyclic precedence graph with reactions as nodes is constructed based on their dependencies. These reactions are then assigned priorities such that any reaction that depends on another reaction always has a lower priority.

The simplest sequential execution then 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 physical time, represented as the number of nanoseconds elapsed since January 1, 1970.
2) Peek at the event queue and wait until physical time matches or exceeds the earliest timestamp on the queue.
3) Pull all events from the event queue that have the same earliest timestamp, find the reactions that these events trigger, and put those reactions on the reaction queue, sorted by priority.
4) Advance logical time to match that earliest timestamp.
5) Execute reactions sequentially in order of priority from the reaction queue. These reactions may produce outputs, which results in more reactions 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`, then an event will be put on the event queue, not the reaction queue.
6) When the reaction queue is empty, go to 2.

This sequential execution has no parallelism and does not support injection of asynchronous events. To support asynchronous events, the framework has to provide a thread-safe `schedule` function that can, for example, be invoked in response to an external interrupt. Our multi-threaded execution engine provides such a thread-safe `schedule` function. It then makes some modifications to the above strategy. First, the wait in step 2 may be interrupted by an asynchronous injection of a new event into the event queue. When that interrupt occurs, after the new event(s) have been injected into the event queue, step 2 is restarted. Second, a pool of worker threads is created (typically with one thread per core) to execute reactions. Each worker thread acquires a single global lock and peeks at the highest priority reaction on the reaction queue. If that reaction is "ready to execute," it transfers the reaction to an **executing queue**, releases the lock, and begins executing the reaction. When the reaction finishes executing, it removes it from the executing queue. A reaction is "ready to execute" if there is no reaction on which it depends, directly or indirectly, (as determined by the acyclic precedence graph) currently on the executing queue. Finally, step 6 is modified so that returning to step 2 occurs only when *both* the reaction queue and the executing queue are empty.

Determining when a reaction is "ready to execute" is tricky and potentially expensive. Conservative approximations, however, may be very inexpensive. The first conservative approximation that we implemented was simply to use the acyclic precedence graph to determine a level, or depth, for each reaction. A reaction that depends on no others has depth zero. A reaction that depends only on reactions with depth zero has depth one, and so forth. With this strategy, a very simple conservative approximation is that a reaction is ready to execute if there are no reactions on the executing queue with smaller depth. We are in the process of developing less conservative approximations that can exploit more parallelism without sacrificing determinacy, but those will need to be reported in a subsequent paper. Their cost of implementation will inevitably be higher.

The above strategy is only able to execute reactions in parallel if they are logically simultaneous. However, in principle, it is possible to execute non-simultaneous reactions in parallel if they have no dependency on one another or if the dependency is known to span enough logical time. Exploiting this additional parallelism is also a work in progress.

For all of these execution policies, there will typically be more than one reaction that is ready to execute at any given time without sacrificing determinacy. Although our current implementation does not do so, a scheduling policy based on rate monotonic and earliest deadline first (EDF) principles can be used to prioritize these executions. The acyclic precedence graph can assign a "rate" to any reaction that is triggered directly or indirectly by a periodic clock. To implement EDF, the deadlines that can be associated with reactions come into play. Not only do these reactions have deadlines, but so do the reactions that they depend on. These dependencies are statically available, so assigning deadlines is straightforward.

Although we have not yet optimized code nor conducted detailed performance measures, our regression tests include programs that execute up to 23 million reactions per second on a single core of a 2.6 GHz Intel Core i7 running macOS Mojave. These are very simple reactions, so this suggests that the overhead is no larger than 43 nanoseconds per reaction invocation. The test suite also includes a multithreaded test case that achieves a speedup of about 3.7 when using four cores compared to one core, suggesting that near linear speedup can be achieved for programs with sufficient parallelism.

IV. Conclusion

We have described a lightweight, low overhead, deterministic, concurrent, and real-time execution platform for software components that communicate via events. There is a great deal of work to be done to efficiently exploit more parallelism, to optimize schedules for real time, and to develop a concise syntax for scalable concurrent operations such as scatter-gather and map-reduce.

References

[1] M. Lohstroh and E. A. Lee, "Deterministic actors," in *2019 Forum for Specification and Design Languages (FDL)*, Sep 2-4 2019, Conference Proceedings, pp. 1–8.

[2] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee, "Actors revisited for time-critical systems," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, pp. 152:1–152:4.

[3] Y. Zhao, E. A. Lee, and J. Liu, "A programming model for time-synchronized distributed real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, Conference Proceedings, pp. 259 – 268.

[4] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *DAC '07: Proceedings of the 44th annual conference on Design automation*. New York, NY, USA: ACM, 2007, pp. 264–265.