Towards Sparse Synchronous Programming in Lua

John Hui
j-hui@cs.columbia.edu
Columbia University
New York, New York, USA

Stephen A. Edwards
sedwards@cs.columbia.edu
Columbia University
New York, New York, USA

ABSTRACT

Most software considers timing a performance issue, but for many embedded applications, the timing of a result is as important as its value. Most modern computers do have precise hardware timers, but they are not easily used to make a whole system timing-aware.

Earlier, we presented the Sparse Synchronous Model for specifying deterministic, concurrent, timing-aware systems and proposed an awkward-to-use C library; here, we present lua-ssm, a Lua library that provides the benefits of SSM in a more accessible setting.

Relying on Lua's incremental garbage collector and support for coroutines, lua-ssm is both easier to use and was simpler to implement than its C counterpart. It provides both a flexible way for users to construct SSM systems and a way for us to more quickly experiment with new features.

CCS CONCEPTS

• **Computer systems organization** → **Real-time languages**; **Real-time system specification**.

KEYWORDS

real time systems, concurrency control, computer languages, timing

ACM Reference Format:
John Hui and Stephen A. Edwards. 2023. Towards Sparse Synchronous Programming in Lua. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 09–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3576914.3587502

1 INTRODUCTION

Earlier, we introduced the Sparse Synchronous Model (SSM) [6, 9], an imperative programming model featuring precise timing prescriptions and deterministic concurrency. We intended for SSM to be the basis of a compiled language that runs on microcontrollers.

However, implementing a new programming language from scratch is difficult. Writing a compiler is laborious and error-prone, and new languages suffer from a lack of libraries and tooling that established languages enjoy. Building and maintaining a custom language runtime is further complicated by our desire to support a wide range of microcontroller platforms.

This work was supported by the NIH under grant RF1MH120034-01.

CPS-IoT Week Workshops '23, May 09–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0049-1/23/05…$15.00
https://doi.org/10.1145/3576914.3587502

```lua
local ssm = require("ssm")

function ssm.pause(d)
  local t = ssm.Channel {}
  t:after(ssm.msec(d), { go = true })
  ssm.wait(t)
end

function ssm.fib(n)
  if n < 2 then
    ssm.pause(1)
    return n
  end
  local r1 = ssm.fib:spawn(n - 1)
  local r2 = ssm.fib:spawn(n - 2)
  local rp = ssm.pause:spawn(n)
  ssm.wait { r1, r2, rp }
  return r1[1] + r2[1]
end

local n = 10

ssm.start(function()
  local v = ssm.fib(n)

  print(("fib(%d) => %d"):format(n, v))
  -- prints "fib(10) => 55"

  local t = ssm.as_msec(ssm.now())
  print(("Completed in %.2fms"):format(t))
  -- prints "Completed in 10.00ms"
end)
```

Figure 1: A synchronous Fibonacci example in lua-ssm with delayed table assignment (`after`), waiting for table writes (`wait`), and concurrent function calls (`spawn`). Library primitives are highlighted in blue; `--` starts comments.

In this work, we implement SSM as a library for Lua, a lightweight scripting language that can be easily embedded in other applications [10]. Our library, lua-ssm[1], extends Lua with synchronous concurrency primitives à la SSM. Lua-ssm builds on existing Lua features like stackful coroutines [4] and *tables* (what Lua calls associative arrays [3]) to implement SSM concepts like processes and scheduled variables. Lua-ssm is implemented in <1000 LoC of pure Lua, does not require modifications to its host language or its runtime, and is compatible with Lua 5.1 to 5.4.

[1]Source code available at https://github.com/ssm-lang/lua-ssm

1.1 Example: Synchronous Fibonacci

Figure 1 shows an adaptation of the Fibonacci example from Edwards & Hui [6], implemented using lua-ssm (imported in line 1). This deterministic synchronous program logically terminates in 10 ms. Like SSM, lua-ssm programs reason about *logical time* (instead of *physical* "wall-clock" time), which only advances when the program explicitly requests it to. Most statements execute and terminate in the same logical time *instant*, and the future is always referred to relative to the current instant. Isolating logical from physical time like this allows our library to give deterministic guarantees about programs' logical temporal behavior, independent of platform speed. The lua-ssm runtime strives to keep logical time synchronized to physical time, but may lag behind.

Our program defines a helper function, `pause(d)` (lines 3–7), which suspends execution for `d` ms using a local *channel table* `t` created by the `Channel` constructor in line 4. The `after()` method on line 5 schedules a *delayed assignment* of `t.go = true` for `d` ms in the future (here, the choice to assign the key `"go"` the value `true` is arbitrary). The call of `after()` is non-blocking, so `pause()` in line 7 immediately calls `wait()` to suspend the process until `t` is written `d` ms later.

The recursive `fib()` function (lines 9–19) uses `pause()` in two ways. In line 11, `fib()` calls `pause()` *synchronously*, meaning the caller, `fib()`, will block until the callee, `pause()`, returns. So, `fib()` will pause for 1 ms when called with `n < 2` before returning `n`.

By contrast, `fib()` calls `pause()` *concurrently* in line 16 using our library's `spawn()` function. This spawns a new *process* that immediately executes the *first instant of* `pause()`. Once the newly spawned `pause()` process reaches the `wait()` at line 6, `fib()` will resume execution at line 17, in the same instant it called `spawn()`.

The two recursive `fib()` calls in lines 15 and 16 are similarly concurrent. A spawned process is placed at a priority just above the process that spawned it and below any existing higher-priority processes, so processes spawned from the same process are prioritized in the order they are created with highest first. So, after line 17, the processes are prioritized as follows (here, $\prec$ means "is at a higher priority than"):

$$\texttt{fib(n - 1)} \prec \texttt{fib(n - 2)} \prec \texttt{pause(n)} \prec \texttt{fib(n)}$$

Lua-ssm uses *tag-range relabeling* [2, 5] to dynamically allocate priority numbers; see Section 3.1.

Functions called synchronously behave as usual: calling a function synchronously blocks its calling processes until the result is calculated and returned. For example, `fib()`'s return value is bound to local variable `v` in line 24 once `fib()` terminates. By contrast, spawning a process returns a *return channel* that will be written with the return value when the spawned process terminates. The return channels from the processes spawned in lines 14–16 are bound to variables `r1`, `r2`, and `rp`. When `fib()` terminates, its return value is written to key `1` of its return channel; e.g., as read in line 18.

Processes can `wait()` on return channels just like they can on regular channel tables. For example, at line 17, `fib()` waits for the spawned `fib()` and `pause()` processes to complete. When invoked with braces, such as in line 17, `wait()` is *conjunctive*—it will only unblock once `r1`, `r2`, and `rp` have *all* been written; see Section 2.2. After `wait()` at line 17, `fib()` returns the sum of the return values from its two concurrent recursive calls.

In lines 23–32, the program uses lua-ssm's `start()` function to call `fib()` within a *synchronous context*. This execution context is managed by lua-ssm and provides processes, priorities, and instants. Library primitives like `wait()` and `spawn()` only work within such a context.

The program in Figure 1 specifies that it executes in exactly 10 ms of logical time because it prescribes that `fib(n)` produces a result at n ms. `start()` executes its first argument, a Lua closure, starting at instant 0. The synchronous call to `fib()` in line 24 blocks until `fib()` returns, so the call to `now()` in line 29 tells us the amount of logical time elapsed since `fib()` was called. Here, all 177 processes are spawned at instant 0, so the overall execution time is set by the largest `n` passed to `fib()`. Though spawning so many processes like this is contrived and inefficient, lua-ssm supports a practically unbounded number of processes. Lua's garbage collector automatically reclaims unused memory after processes terminate.

1.2 Overview

In this paper, we discuss the design and implementation of lua-ssm. In Section 2, we describe our library's API and semantics; in Section 3, we discuss our library's implementation; in Section 4, we conclude with a discussion on related and future work.

2 SEMANTICS

Lua-ssm adapts SSM to better suit the idioms of Lua. In this section, we describe the semantics of our library and compare it to the SSM toy language of Edwards & Hui [6]. We will focus our discussion on library-provided primitives like `after()` and `wait()`. Utilities like `pause()` can be implemented using these primitives, and can be made available to lua-ssm programs in a separate library.

2.1 Channel Tables

Lua-ssm's channel tables replace SSM's scheduled variables, and support delayed assignments (using `after()`) and blocking (using `wait()`). They are implemented as an extension of Lua's native tables, which consist of entries that map non-`nil` keys to non-`nil` values. Their fields are initialized by passing a table literal to the `Channel` constructor, and are accessed using Lua's dot- and index-notation:

```
local tbl = ssm.Channel { key = 42 }
assert(tbl.key == 42)      -- dot notation
assert(tbl["key"] == 42)   -- equivalent index notation
```

Assigning to keys in a channel table is an *instantaneous assignment*: in contrast to assignments scheduled using `after()`, instantaneous assignments take effect immediately like regular table writes:

```
tbl.key = 420      -- write existing entry
tbl.newkey = 0     -- define new entry
assert(tbl.key == 420 and tbl.newkey == 0)
```

As in SSM, instantaneous assignments unblock all *lower priority* processes waiting on that channel table. However, lua-ssm's channel tables are written at a finer granularity than SSM's scheduled variables, whose scheduled variables can only be written in their entirety. In fact, there is no direct lua-ssm equivalent for updating an entire table: `tbl = different_tbl` sets `tbl` to refer to `different_tbl`, rather than writing values into the channel table referred by `tbl`.

Lua-ssm's per-key semantics extend to its `after()` primitive, which schedules delayed assignments only to the specified keys:

```
tbl:after(ssm.msec(10), { key = 24 })
ssm.wait(tbl)  -- wait for the write; only tbl.key is written
assert(tbl.key == 24 and tbl.newkey == 0)
```

Note that the delayed assignment `key = 24` is specified as a table literal in the second argument to `after()`; we can schedule multiple assignments for the same time by adding more entries to this table.

Lua-ssm also allows us to schedule delayed assignments to multiple keys at *different times*, provided those keys do not overlap:

```
tbl:after(ssm.msec(10), { key = 10 })
tbl:after(ssm.msec(20), { newkey = 20 })
ssm.wait(tbl)  -- wait for the write; only tbl.key is written
assert(tbl.key == 10 and tbl.newkey == 0)
ssm.wait(tbl)  -- wait for the write; this time tbl.newkey is written
assert(tbl.key == 10 and tbl.newkey == 20)
```

Each key may only have one outstanding assignment; scheduling another assignment on the same key overwrites any existing one.

2.2 Conjunctive and Disjunctive Waiting

As in SSM, lua-ssm's `wait()` primitive allows processes to suspend execution until one or more channel tables are written. That write may be due to any delayed assignment, or an instantaneous assignment by a higher-priority process. `wait()` is lua-ssm's only directly blocking primitive; processes may also block indirectly by synchronously calling a function that calls `wait()`.

Any write to a channel table triggers a `wait()`, including those that do not change a value:

```
local tbl = Channel { go = true }
tbl:after(ssm.msec(10), { go = true })
ssm.wait(tbl) -- unblocks after 10ms
```

The "**wait**" primitive in the original SSM proposal was strictly *disjunctive*: **wait** $a \mid b$ would unblock the instant *either* a or b are updated. *Conjunctive* waiting—unblocking when *both* a and b are updated—using SSM's conjunctive **fork** primitive, which spawns multiple processes and blocks until all of them terminate:

$$\textbf{fork}\ (\textbf{wait}\ a)\ \&\ (\textbf{wait}\ b)$$

However, SSM's reliance on this **fork-wait** pattern makes it awkward to express statements such as, "wait until both buttons (e.g., a and b) have been pressed, or we've timed out (e.g., on t)." The top-level disjunction needs to be distributed into the inner conjunction:

$$\textbf{fork}\ (\textbf{wait}\ a \mid t)\ \&\ (\textbf{wait}\ b \mid t)$$

Further work is needed to determine which condition—both buttons being pressed or timing out—caused the statement to unblock.

To address this limitation, lua-ssm's `wait()` provides both disjunctive and conjunctive semantics, with the unblocking condition given in disjunctive normal form (DNF). Its general form is `wait(`w_1`, ..., `w_n`)`, where each *wait specification* w_k may be a single channel table c or a Lua array of channel tables `{`c_1`, ..., `c_k`}`. `wait()` is disjunctive over all wait specifications, but conjunctive within each wait specification, so the above unblocking condition can be succinctly expressed as:

```
ssm.wait({a, b}, t) -- i.e., (a & b) | t
```

Since Lua syntax allows us to omit parentheses for function calls when the only argument is a table literal, we can write purely conjunctive `wait()` statements like `wait {r1, r2, rp}`, from line 17 of Figure 1. `wait()` returns an array of Booleans indicating which wait specifications were met.

2.3 Concurrent Function Calls

Within a synchronous context, all running processes are ordered according to their *priority*—we write $p \prec p'$ to mean process p has a higher priority than process p'. This total ordering is what gives SSM its determinism: each process may execute at most once per instant, and once a process suspends execution, no process of a higher priority may execute that instant.

Lua-ssm provides two concurrent function call primitives for creating processes, `spawn()` and `defer()`. When a process is created, the user does not need to explicitly specify a process priority. Instead, the priority of a newly created process—the *child*—is determined relative to that of the current running process—the *parent*. When a child is `spawn()`ed, it is given the next highest priority, i.e., the lowest priority that is still higher than that of its parent. The child immediately runs its first instant before yielding control to its parent. `defer()` is the dual of `spawn()`: a child created by `defer()` is given the next lowest priority. It does not run its first instant until the parent suspends or terminates.

The first argument of `spawn()` and `defer()` is a *synchronous function* that the newly created process will run. A synchronous function is a Lua function (or closure) that may invoke lua-ssm primitives, and can only run within a synchronous context. Since anonymous closures are first-class values in Lua, we can pass them directly to `spawn()` and `defer()`. For example, the following helper routine waits on `chan` with a specified `timeout`, using a closure:

```
local function wait_for(chan, timeout)
  return ssm.wait(chan, ssm.spawn(function()
    local t = ssm.Channel {}
    t:after(timeout, { go = true })
    ssm.wait(t)
  end))
end
```

The closure's body is the same as the `pause()` implementation from Figure 1, except it captures `timeout` from its enclosing scope.

If we want to run `wait_for()` as its own process, we can pass its name as the first parameter, followed by the arguments:

```
ssm.spawn(wait_for, some_chan, some_timeout)
```

For convenience, lua-ssm also supports concurrent function calls using *method call* syntax (e.g., `ssm.fib:spawn()`, as seen in Figure 1) when synchronous functions are defined on keys of the `ssm` module:

```
function ssm.wait_for(chan, timeout)
  -- same as wait_for()
end
```

When a synchronous function is defined this way, concurrent calls may be made with method call syntax:

```
ssm.wait_for:spawn(some_chan, some_timeout)
```

```
local ctx = {
  time = 0,           -- the current time
  proc = nil,         -- the current running process
  event_q = Heap {},  -- pending delayed assignments
  run_q = Heap {},    -- processes to execute this instant
}
```

Figure 2: Initialization of lua-ssm's synchronous context.

2.4 Synchronization and Return Channels

The assigned priorities are relative to those of processes that exist at the time of creation. So, calling `spawn()` multiple times will create processes with successively lower priorities:

```
ssm.foo:spawn() -- highest priority
ssm.foo:spawn() -- next highest priority
-- (parent) lowest priority
```

and vice versa for `defer()`:

```
ssm.foo:defer() -- lowest priority
ssm.foo:defer() -- next lowest priority
-- (parent) highest priority
```

Lua-ssm's concurrent function calls differ from SSM's blocking **fork** in that they do not suspend the execution of the parent. As such, they allow more flexibility than nested fork/join. For instance, lua-ssm permits calling `spawn()` in a loop to create a variable number of processes in one instant.

A lua-ssm process is created with a *return channel* that allows a parent to wait for its children and receive their return values. Return channels behave like regular channel tables, and are (instantaneously) assigned the return values of a process when it terminates. As such, they function as a *future* for the child process, which can be queried by the parent to check for return values.

Return channels support Lua functions that return multiple values at once (e.g., `return true, "mesg"`). These are assigned to return channels positionally, so each return value is written to a successive key, starting from 1:

```
local rc = ssm.foo:spawn()
-- foo(): return true, "mesg"
assert(rc[1] == true and rc[2] == "mesg")
```

3 IMPLEMENTATION

As a Lua library, lua-ssm takes advantage of its host language's existing features, libraries, and ecosystem. For instance, Lua's incremental garbage collector relieves a major burden from the implementation of SSM, while its standard library, module system, and FFI capabilities allow lua-ssm to integrate with existing code.

However, SSM's scheduled variables and synchronous execution model are foreign concepts to Lua, as they rely on the synchronous context maintained by lua-ssm. The synchronous context, shown in Figure 2, keeps track of the timestamp of the current instant, the current running process, and two queues used by the scheduler.

In this section, we discuss how lua-ssm uses Lua's coroutines and metatables to embed a synchronous programming model within a procedural scripting language.

```
local function run_instant()
  ctx.time = next_scheduled_event_time(ctx)

  for c in scheduled_events(ctx.event_q, ctx.time) do
    channel_do_update(c)
  end

  for p in scheduled_processes(ctx.run_q) do
    process_resume(p)
  end
end
```

Figure 3: lua-ssm's "tick" function, which executes the synchronous context for an instant. Each backend drives execution by calling `run_instant()` while synchronizing with a platform-specific clock. `channel_do_update()` (not shown) copies scheduled updates from each channel's `later` field to its `shadow` field and adds sensitive processes to `ctx.run_q`.

```
local function process_resume(next_proc)
  local prev_proc
  prev_proc, ctx.proc = ctx.proc, next_proc
  coroutine.resume(ctx.proc.tid)
  ctx.proc = prev_proc
end

function ssm.wait(...)
  local wait_specs = { ... }
  sensitize(ctx.proc, wait_specs)
  while not specs_satisfied(wait_specs) do
    -- keep waiting until at least one wait specification is satisfied
    coroutine.yield()
  end
  return desensitize(ctx.proc, wait_specs)
end
```

Figure 4: Implementation of `process_resume()` and `wait()`, using Lua's built-in `coroutine`.

3.1 Scheduling Processes

Like the runtime proposed by Edwards & Hui [6], lua-ssm uses two priority queues implemented as binary heaps: an event queue of delayed assignments on channel tables and a *run queue* of active processes scheduled in the current instant. Each instant is executed by the "tick" function shown in Figure 3.

Lua's built-in coroutines [4] greatly simplify managing activation records of suspended threads. `process_resume()`, shown in Figure 4, uses Lua's built-in `coroutine.resume()` to resume a suspended process; `wait()` calls `coroutine.yield()` to return to the `coroutine.resume()` call site. Since Lua's coroutines are *stackful*, `coroutine.yield()` (hence `wait()`) works at arbitrarily deep levels of the Lua call stack.

Where possible, lua-ssm also uses Lua's own call stack to avoid adding to the run queue unnecessarily. Synchronous function calls

```
function ssm.spawn(func, ...)
  local retchan = ssm.Channel {} -- allocate return channel

  -- current priority goes after new priority
  local prio = ctx.proc.prio
  ctx.proc.prio = prio:insert_after()

  local proc = process_new(func, { ... }, retchan, prio)
  process_resume(proc) -- run first instant of new process

  return retchan
end

function ssm.defer(func, ...)
  local retchan = ssm.Channel {} -- allocate return channel

  -- new priority goes after current priority
  local prio = ctx.proc.prio:insert_after()

  local proc = process_new(func, { ... }, retchan, prio)
  process_enqueue(proc) -- add deferred process to run queue

  return retchan
end
```

Figure 5: Implementation of `spawn()` and `defer()`.

require no special treatment since they are just regular Lua function calls. `spawn()`, shown in Figure 5, avoids touching the run queue by calling `process_resume()`, instead of yielding back to the tick loop like the original SSM runtime does.

Each SSM process must be assigned a unique priority to ensure that they are totally ordered. The SSM runtime presented by Edwards & Hui assigns priorities to processes according to their position in the process tree, but this scheme limits the depth of the process tree to the number of bits given to priorities (i.e., 32).

To overcome this limitation, lua-ssm implements priorities using Dietz & Sleator's *tag-range relabeling* algorithm [2, 5]. Their solution to the order maintenance problem occasionally redistributes densely clustered priority numbers to avoid saturating the range of assignment numbers. Abstractly, this redistribution is equivalent to performing rotations on the process tree to limit its height, and incurs an amortized $O(\log n)$ cost when inserting a new priority. This algorithm guarantees up to $2^{N/2} - 1$ distinct priorities, where N is the number of bits used to represent integers. Lua uses IEEE 754 double-precision floating-point numbers [1] by default, which theoretically allows lua-ssm to support up to $2^{52/2} - 1$ processes (though Lua will likely run out of memory before then).

In lua-ssm's implementation of priorities, the `insert_after()` method constructs a new priority that is inserted immediately after the priority it is called on. The implementations of `spawn()` and `defer()` in Figure 5 use this method to assign a new priority relative to that of the current running process.

```
function ssm.Channel(init)
  local chan = {
    handle = {},   -- given to user; kept empty
    shadow = {},   -- where entries are actually stored
    later = {},    -- scheduled updates
    triggers = {}, -- what to run when updated
    -- other metadata
  }
  chan.shadow.after = channel_after -- attach after() method
  for k, v in pairs(init) do      -- initialize shadow table
    shadow[k] = v
  end
  chan.__index = chan.shadow      -- read from shadow table
  chan.__newindex = channel_set   -- overload assignment
  setmetatable(chan.handle, chan)
  return chan.handle
end
```

Figure 6: Constructor for channel tables.

3.2 Implementing Channel Tables

Lua-ssm uses Lua's *metatables* mechanism to extend ordinary tables with the capabilities of SSM scheduled variables. Metatables can be attached to other tables using Lua's built-in `setmetatable()`, and retrieved using `getmetatable()`:

```
local t, m = {}, {}
setmetatable(t, m) -- m is used as the metatable of t
assert(getmetatable(t) == m)
```

The `Channel()` constructor, shown in Figure 6, attaches metatables to ordinary tables to hold metadata needed for implementing channel tables, including the times and values of a delayed assignments (`later`), and the list of processes that need to be woken up when that channel table is updated (`triggers`).

A table's metatable can also be populated with *metamethods* that overload certain table operations; lua-ssm overloads the `__index` and `__newindex` metamethods to intercept accesses to table entries:

```
t[k]       -- reads from getmetatable(t).__index[k]
t[k] = v   -- calls getmetatable(t).__newindex(t, k, v)
```

Since these metamethods are only invoked when reading and writing *absent* entries, lua-ssm maintains the actual table entries in a separate `shadow` table. The table returned to the user is just a `handle` and is not used to store entries. This ensures that the overloaded `__index` and `__newindex` are used for every channel table access.

The constructor initializes the `shadow` table using the `init` table. When a user assigns to a channel table's `handle`, the overloaded `__newindex` sets the value in the `shadow` table—leaving the `handle` empty—and schedules sensitive lower-priority processes (in `triggers`) for execution. When a user reads from a channel table's `handle`, its overloaded `__index` forwards the read to the `shadow` table.

3.3 Backend Support

The core lua-ssm library is platform-agnostic and isolates its logical timing semantics from the external environment. This isolation maintains the correct *logical* behavior in spite of the program's

```
1  local ssm = require("ssm") { backend = "luv" }
2
3  function ssm.pause(d)
4    -- same as before, see Figure 1
5  end
6
7  ssm.start(function()
8    local stdin = ssm.io.get_stdin()
9    local stdout = ssm.io.get_stdout()
10
11   while ssm.wait(stdin) do
12     if not stdin.data then -- stdin was closed
13       break
14     end
15     local str = stdin.data -- buffer data from stdin
16     ssm.pause(250)         -- suspend for a bit
17     stdout.data = str      -- write to stdout
18   end
19   stdout.data = nil        -- close stdout
20 end)
```

Figure 7: An "echo" program using lua-ssm's `luv` backend.

physical timing characteristics. Meanwhile, lua-ssm's scheduler is "driven" by a *backend* that is responsible for synchronizing the program with physical time. A backend has access to platform-specific timing and I/O capabilities (e.g., device registers, system calls), and repeatedly invokes the core scheduler by calling `run_instant()` (Figure 3). A backend may expose asynchronous input sources and output destinations to lua-ssm programs as channel tables.

Lua-ssm currently supports two backends. The `simulation` backend simulates the execution of lua-ssm programs without synchronizing to a physical clock. It lacks external dependencies, so it is useful for platform-agnostic prototyping. Due to SSM's determinism, simulation can be used as an oracle for logical behavior.

The second backend, `luv`, uses Lua bindings to libuv[2], a multiplatform asynchronous I/O library. For example, the interactive terminal application in Figure 7 echoes its standard input to standard output after a 250 ms delay. The standard input and output streams are modeled using channel tables (bound to `stdin` and `stdout` on lines 8–9) that can be waited on and written to. A libuv callback for standard input assigns the received input to `stdin.data`, which the echo program reads (line 15), while a low-priority handler process—created using `defer()` by `io.get_stdout()`—forwards assignments to `stdout.data` (line 17) to standard output. The stream is closed when `nil` is assigned to the `data` field (lines 12 and 19).

4 RELATED AND FUTURE WORK

Lua-ssm is not the first implementation of the Sparse Synchronous Model [6]. Hui & Edwards [9] are developing a functional language with user-defined algebraic data types, references, and pattern matching that implements SSM; Krook et al. [11] embed an SSM-based imperative language, Scoria, Haskell. Both of these synchronous languages compile to portable C code that links against Edwards & Hui's `ssm-runtime` library and runs on embedded hardware. While lua-ssm is based on the same programming model, it is implemented entirely in Lua. It overcomes many of `ssm-runtime`'s limitations that stem from its low-level implementation in C.

Like Copilot [13], Haski [14], and Scoria [11], lua-ssm is an *embedded domain-specific language* (eDSL) [8] for synchronous computing, though it uses Lua rather than Haskell as its host language. Unlike the aforementioned eDSLs, which are *deep* embeddings that generate C code, lua-ssm is a *shallow* embedding whose execution takes place within its host language. While a shallow embedding precludes compiling and optimizing lua-ssm programs within its host language, it enables easier integration with the Lua ecosystem.

Like SSM, Lingua Franca (LF) [12]'s *reactor model* is also inspired by discrete-event systems, but LF takes the opposite implementation strategy of lua-ssm. Rather than embedding the reactor model in an existing language, LF embeds fragments of existing languages like C and TypeScript within its model. While LF emphasizes analyzability and scalability, lua-ssm prioritizes flexibility and expressiveness.

We believe that embedded application development with lua-ssm will greatly benefit from Lua's flexibility and FFI capabilities. Using an interpreted language will likely impact performance, so we plan to quantify that impact and replace lua-ssm's hot code paths with optimized C or Pallene [7]. Finally, we hope to evaluate the suitability of using lua-ssm for non-performance-critical tasks such as real-time application prototyping, coordination, and testing.

[2]See https://libuv.org and https://github.com/luvit/luv

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two Simplified Algorithms for Maintaining Order in a List. In *European Symposium on Algorithms*. 152–164. https://doi.org/10.1007/3-540-45749-6_17

[3] Jon Bentley. 1985. Programming Pearls: Associative Arrays. *Commun. ACM* 28, 6 (June 1985), 570–576. https://doi.org/10.1145/3812.315108

[4] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems* 31 (2009), 6:1–6:31.

[5] P. Dietz and D. Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the Symposium on Theory of Computing (STOC)*. 365—-372. https://doi.org/10.1145/28395.28434

[6] Stephen A. Edwards and John Hui. 2020. The Sparse Synchronous Model. In *Forum on Specification and Design Languages (FDL)*. Kiel, Germany. https://doi.org/10.1109/FDL50818.2020.9232938

[7] Hugo Musso Gualandi and Roberto Ierusalimschy. 2020. Pallene: A companion language for Lua. *Science of Computer Programming* 189 (apr 2020), 102393. https://doi.org/10.1016/j.scico.2020.102393

[8] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *Comput. Surveys* 28, 4es (Dec. 1996), 196–es. https://doi.org/10.1145/242224.242477

[9] John Hui and Stephen A. Edwards. 2022. The Sparse Synchronous Model on Real Hardware. *ACM Transactions on Embedded Computing Systems* (Dec. 2022). https://doi.org/10.1145/3572920 Just Accepted.

[10] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *Proceedings of the History of Programming Languages (HOPL III)*. 2–1–2–26. https://doi.org/10.1145/1238844.1238846

[11] Robert Krook, John Hui, Bo Joel Svensson, Stephen A. Edwards, and Koen Claessen. 2022. Creating a Language for Writing Real-Time Applications for the Internet of Things. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Shanghai, China.

[12] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems* 20, 4 (July 2021), 1–27. https://doi.org/10.1145/3448128

[13] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. Copilot: A Hard Real-Time Runtime Monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification (LNCS)*. Springer.

[14] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards secure IoT programming in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. ACM. https://doi.org/10.1145/3406088.3409027