Sparse Synchronous Programming with Temporal Abstractions

John Hui

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2024

Abstract

Sparse Synchronous Programming with Temporal Abstractions

John Hui

For many embedded applications, the timing of a result is as important as its value. However, most programming languages treat timing as a side effect, so they cannot convey temporal behavior without compromising precision, efficiency, or flexibility. This dissertation presents the Sparse Synchronous Model (SSM), a programming model for building temporal abstractions with high-level languages. SSM is deterministic and defines behavior in terms of logical time, but is more expressive than the synchronous languages it was inspired by. It uses the same abstractions for internal events and external I/O, so the meaning of each program is preserved across different platforms. The main contributions of this work are the formal semantics of SSM, and SSLang, a standalone functional programming language that implements SSM and runs on microcontrollers. SSLang's runtime keeps the software synchronized with the real world, and uses timing-aware hardware peripherals to achieve jitter-free I/O with sub-microsecond precision. The dissertation also describes two embeddings of SSM in existing languages to show that the model is compatible with conventional programming paradigms. Together, these results illustrate the value of extensible, timing-aware programming abstractions for building reliable real-time systems.

Table of Contents

List of Figures

Acknowledgments

Stephen A. Edwards could not have been a better advisor to me. He encouraged me to look into research problems driven by curiosity and obsession. He taught me to write about my findings with illustrative examples, clear language, and just enough personality to keep it fun. He always had something witty to say whenever I looked to him for advice, and punctuated my PhD with a series of memorable quotes. My thesis would not exist without his guidance.

I would also like to thank Martha Kim, Luca Carloni, Mark Santolucito, and Jeronimo Castrillon for serving on my thesis defense committee.

Martha and Luca were also on my candidacy exam and thesis proposal committees. I'm grateful to them for always being there for me and making me feel welcome in the department.

Mark led an informal reading group on category theory that was both stimulating and perplexing. Determining its impact on this dissertation is left as an exercise to the reader.

Edward A. Lee agreed to be on my defense committee but ultimately could not attend due to scheduling reasons. His work on real-time programming models deeply influenced my thesis.

Alan Jeffrey's brilliance and mentorship made me realize how much I still have to learn and what I want to do in the years following my PhD. This dissertation is dedicated to his memory.

Arjun Guha gave me the encouragement to continue doing research when I was at my lowest point. I'm appreciative of the SIGPLAN-M organizers for putting us in touch.

Many collaborators contributed to the work presented in this dissertation. I point out their names and contributions in each chapter, but here I want to highlight the central role Feitong Qiao and Emily Sillars played in building up the "SSLang gang" at Columbia.

Robert Krook is a wonderful friend and collaborator. He opened my eyes to the virtues of embedded DSLs, property-based testing, and Swedish candy—thanks for all the Fish!

Jason Yuan gave me advice on fonts, and assured me that my taste is not as bad as I fear.

Maybe the real thesis is all the friends we made along the way. If this sentiment was taken literally my dissertation would look very different and have chapters dedicated to Andreas Kellas, Clayton Sanford, David Williams-King, Martha Barker, Maxwell Levatich, and Vikram Nitin, all companions on my CS PhD journey at Columbia who have had a positive impact on me.

The SIGPLAN AV team was both an outlet for me and a way to give back to the research community; I'm glad that Guilherme Espada, Apoorv Ingle, and Benjamin Chung got me involved. Some surprising points were made in their feedback on this dissertation.

Jae Woo Lee indirectly convinced me to pursue a PhD by helping me discover my passion for computer systems and teaching. Most of what I know about systems programming stems from my experiences as his teaching assistant.

Hans J. Montero came up with the name "SSLang" and exposed me to many novel phrases, some less cringe than others. He had an epic role in shaping the community around Jae's army of teaching assistants, which I am grateful to have been part of even as a former AP TA.

Mona Ye's celebrated work in uplifting and encouraging me has been a constant source of motivation to write, write, and *write*. Thanks for being patient and putting up with my silliness.

I'm indebted to my parents Jing Zhao and William Hui for their unwavering love and support. Gloria Hui had asked to come first in the acknowledgments, but seemed ok with coming last. She thinks I spent too long on this and just wants me to say some good words about her: great sister.

Dedication

To Alan Jeffrey.

John Hui

Chapter 1: Introduction

Consider the sleep routine,[1] which suspends execution for some specified duration of time. We can use sleep to implement a reaction time game, where the player must press a button as soon as they can react to the blink of an LED light:

```
reactionTime button led duration = do
    sleep duration
    blink led (msecs 500)
    measureDuration (waitFor button Press)
```

Here, sleep pauses the game for the given duration, but it is not the only routine to prescribe timing behavior: blink turns the led on, then off 500 ms later; waitFor suspends execution until the button is Pressed; measureDuration computes the time elapsed while waiting.

Temporal abstractions such as these encapsulate timing behaviors that can be combined to implement more complex behaviors. For instance, reactionTime is itself an abstraction for just one round of a game that might run for several rounds. Temporal abstractions are building blocks for *real-time reactive systems* [8], which are used to implement timing-sensitive applications[2] like video games and robotics where the timing of a computation is as important as its value. Such systems are characterized by programs that react to inputs like button presses and emit outputs like LED on/off signals.

My reactionTime example illustrates several ways in which real-time reactive systems can be under-specified. How precise is sleep? Does the system wait 500 ms for the LED to blink, or does it somehow toggle the LED in the background? And when does measureDuration start its timer, relative to the waitFor expression?

[1] I certainly gave sleep routines much thought as a graduate student.

[2] In contrast to timing-*insensitive* applications, where only the computed value matters, or *performance*-sensitive applications, where faster is "better" but not necessarily "more correct" [44].

These questions motivate the thesis of my dissertation, which is about the synergy between temporal abstractions and programming languages in clarifying these issues:

> *Temporal abstractions can be built using programming languages; programming languages can harness their platform's timing capabilities through temporal abstractions.*

The first part of my thesis may seem self-evident for readers who recognize languages as a tool for creating and manipulating abstractions. However, conventional programming languages assume timing is not a condition of correctness, and treat it as an opaque implementation detail. While this assumption makes it easier to argue that a faster, optimized implementation is "just as correct" as one that is slower but sound, it leads to vague and imprecise timing guarantees. As a result, programmers are impelled to forego high-level software abstractions in favor of low-level, non-portable hardware interfaces whose timing behavior can be better understood. My thesis advances an alternative vision where languages help us rather than hinder us, providing us an expressive framework to build complex high-level programs from abstractions with precise and reliable timing behavior.

The latter part of my thesis is an instantiation of the idea that when languages "say more" their implementations can "do better" [18]. High-level temporal abstractions emphasize *what* the behavior should be, and leave more opportunities for the compiler and runtime to discover *how* to best implement that behavior. My dissertation focuses on microcontroller platforms, which are widely used in real-time embedded systems. Despite their limited processor speed and memory, microcontrollers are well-suited for time-sensitive applications because their simplicity makes timing easy to predict.[3] However, they only expose low-level, platform-specific mechanisms for controlling and measuring time, which can be difficult to integrate with timing-oblivious languages in a way that is idiomatic and robust. My thesis proposes using temporal abstractions to encapsulate those mechanisms, to provide a robust and portable programming framework for real-time applications.

[3] Their low cost and power efficiency are also desirable qualities for robotics and other cyber-physical systems.

To support my thesis, my dissertation presents the *Sparse Synchronous Model* (SSM), a reactive programming model designed to bring precise and deterministic temporal abstractions to higher-order programming languages [31, 53]. SSM considers timing as important as data, and features concrete, portable timing prescriptions (e.g., saying "30 ms" instead of "15 ticks"). In contrast to traditional synchronous languages and real-time operating systems, SSM is designed for sporadic workloads, characterized by extended periods of inactivity punctuated by occasional bursts of work.[4] The model is called "sparse" because its synchronous execution model is driven by intermittent events rather than a periodic timer, and supports advancing time by arbitrary increments between discrete instants of computation.

My dissertation examines several aspects of sparse synchronous programming. The rest of this chapter is devoted to motivation, background, and related work. In Chapter 2, I present the programming model and define its formal semantics using a call-by-value lambda calculus that illustrates novel aspects of SSM. In Chapter 3, I explore the language design space by comparing several incarnations of the programming model, including SSLang, a standalone functional programming language based on SSM. In Chapter 4, I walk through the implementation of a language runtime for SSM that leverages microcontroller hardware to achieve sub-µs precision. In Chapter 5, I describe several experiments that quantify SSLang's performance. Chapter 6 outlines future work and concludes my dissertation.

This dissertation describes joint work with my advisor Stephen A. Edwards and other collaborators. I use collective first-person pronouns (e.g., "we") when stating our results and observations; singular first-person pronouns (e.g., "I") are used for narration and personal remarks.

[4] Such workloads are also representative of my time as a graduate student.

```
sleep (usecs 150) -- sleep for 150 μs
sleep (usecs 150) -- snooze for 150 μs
```

(a)

```
sleep (usecs 100) -- sleep for 100 μs
sleep (usecs 100) -- snooze for 100 μs
sleep (usecs 100) -- and another 100 μs
```

(b)

Figure 1.1: Two programs that run for roughly 300 μs.

1.1 Motivation

Conventional programming languages such as C, Lua, and Haskell are based on what Harel and Pneuli call *transformational* models of computation [44]: they compute output data from input data without concern for the passage of time. These models—and the languages based on them—are suitable for applications such as compilers and batch data processing tasks, where it is more pragmatic to treat execution time as an emergent property of the implementation, rather than as a requirement imposed by the programming model. While transformational languages and models can express rich abstractions for *control* and *data* [50], they lack the vocabulary for precise and reliable *temporal* abstractions.

Strictly speaking, it is not possible to implement temporal abstractions like sleep entirely within a language like C—it is "just as correct" for a C function to return in 12 μs as it is to run on for 12 years [9]. Instead, C programs rely on platform-specific mechanisms, like hardware alarms on microcontrollers and blocking system calls on time-sharing operating systems, to achieve the desired temporal behavior. Many timing-oblivious languages provide standard libraries which expose those low-level mechanisms as *timing primitives*, but those languages are semantically unaware of the effect such primitives have on timing behavior.[5]

Programming languages—even timing-oblivious ones—provide a framework for building temporal abstractions from the *composition* of other abstractions. For instance, Figure 1.1a shows a program that runs for roughly 300 μs[6] by sequencing two statements which each sleep for 150 μs.

[5] C and C++ also provide a `volatile` keyword to ensure the compiler does not optimize away the load and store instructions needed to interface those low-level timing mechanisms (e.g., for reading and writing memory-mapped timer registers). However, this keyword is easily misunderstood by both compiler writers and users alike [32], and say nothing of timing.

[6] I qualify this with the word "roughly" to avoid any pretense of precision.

This implementation is rather contrived, but begs the question: is its behavior equivalent to the program in Figure 1.1b, which sleeps 100 μs thrice?

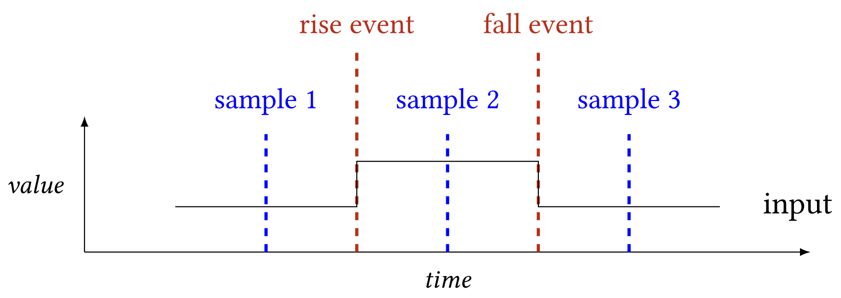
Most languages do not specify any timing constraints for sequential composition, so the actual execution time of each program depends on how sequential composition is implemented. This ambiguity leaves us unable to rigorously reason about their temporal behavior without making assumptions about how they are compiled or interpreted. In fact, execution time is also highly sensitive to the architecture and state of the underlying hardware, and is incredibly difficult to predict even when relevant factors of the system are fixed and known [98]. This difficulty is only compounded by timing-oblivious language features like functions and objects, whose functionality is well-understood but whose implementation may be complex and opaque.

1.2 Synchrony

For many languages, decoupling their semantics from timing is a good idea because it liberates language implementers from strict timing requirements that may compromise the soundness of compiler and runtime optimizations. In the 1980s, French researchers seeking to do the same for reactive systems developed the *synchronous* languages Esterel, Lustre, and Signal [9, 14, 41, 79], which specify behavior in terms *logical* time instead of *physical* time.[7] Inspired by synchronous digital circuits, these languages perform computations along the ticks of a (logical) global clock. Programs thus experience time as a succession of discrete ticks, called *instants*.

The defining characteristic of instants is that they are *logically* instantaneous: the fiction of an infinitely fast processor[8] greatly simplifies reasoning about temporal behavior [8]. In other words, synchrony encapsulates execution time to ensure the semantics of synchronous programs are independent of platform speed. To have synchronous programs behave as if execution were instantaneous, it is the implementation's responsibility to keep logical and physical time in sync. Synchronous languages also require that computation within each instant always terminates; otherwise, logical time would not be able to advance.

[7] Also called "model" time versus "wall-clock" time.
[8] Berry calls this the "synchrony hypothesis" [11].



(a) How sample- and event-driven systems react to an input signal.

```
forever
    if inputs changed
        call Tick
    pause
```

(b) Emulating events through by sampling inputs. The system repeatedly checks for input changes, which can be expensive.

```
forever
    call Tick
    wait clock event
```

(c) Emulating sampling with a periodic clock signal. The machinery for event handling may reduce overall throughput.

Figure 1.2: Sample-driven systems versus event-driven systems.

1.3 Real-Time Execution Models

An important aspect of real-time reactive systems is how they determine *when* should computations happen. Their execution models can be categorized according to the two paradigms illustrated by Figure 1.2a: *sample-driven* systems are purely time-triggered and periodically sample inputs from their environment, whereas *event-driven* react to input events like value changes. Samples and events are what trigger logical instants of computation in synchronous systems.

As shown in Figures 1.2b and 1.2c, sample- and event-driven systems can emulate each other but make trade-offs that favor certain workloads. The sample-driven approach works well for constantly evolving systems by supplying a steady “heartbeat” for the program to periodically interact with its environment. Some familiar examples of suitable applications include physics simulations and audio processing, which break down continuous functions into discrete, periodic time steps. However, sample-driven systems evaluate ticks unnecessarily for applications with sparse, irregular workloads. While reducing the clock frequency improves the efficiency of long suspensions, doing so comes at the cost of timing precision, and may not be a desirable trade-off when the system needs to perform occasional but intensive bursts of work.

SSM uses an event-driven paradigm that instead favors sporadic, aperiodic events. Discrete-event systems remain idle in the absence of events, which is more efficient when events are infrequent. However, this approach typically introduces some overhead (e.g., an event queue) that reduces the overall throughput of the system when subjected to intensive workloads. Compared to sampling, events give programmers more control over when computations happen, but takes away a global heartbeat with which programs can measure and advance time. Programs may generate and react to *internal events* to drive execution without external stimulus, but leaves the programmer with the additional burden of managing such events.

1.4 Discrete Events

Unlike SSM, many event-driven models are not designed for logically precise timing. For example, libuv [85], an I/O framework used by Node.js, runs on an event loop that handles a single event at a time and does not support the notion of logically simultaneous events [40, 73]. Programming frameworks based on *asynchronous* events avoid the hazards of multithreading by dispatching concurrent computations from a single-threaded event loop, but those events do not reliably convey timing.

Instead, SSM is based on the foundations of discrete-event simulations [61, 64], and associates events with the discrete, synchronous instants. Discrete-event systems typically characterize events as value-timestamp tuples: we write $v@t$ to denote a value v "tagged" by a timestamp t.[9] Timestamps are totally ordered: we write $t < t'$ (or $e < e'$) to say that a timestamp t (or event e) is "earlier" than another distinct timestamp t' (or event e').

SSM is hardly the first programming model inspired by discrete-event simulation. Other discrete-event systems [12, 70, 89, 99] make different trade-offs between programmability, analyzability, and flexibility; they are discussed in Section 1.7. Discrete events also underlie the hardware description languages Verilog [55] and VHDL [56], widely used to simulate, test, and synthesize digital hardware.

[9] A timestamp is sometimes also called a "tag" in related literature [64].

```
wait click
timeout (msecs 500) (wait click)
clicked ← justHappened click
if clicked
    then singleClick
    else doubleClick
```

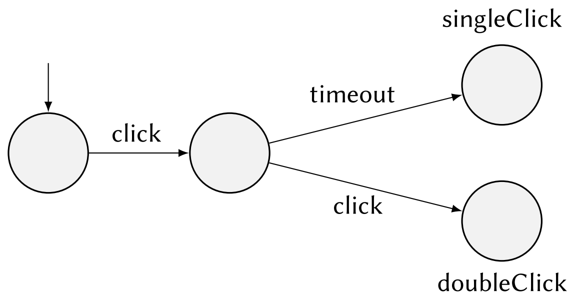


Figure 1.3: An example using control flow to encode state, for handling a double click event.

1.5 Dataflow and Control Flow

The synchronous and discrete-event languages share common notions of time, but they embody different programming paradigms. For instance, Lustre [41] and Signal [8, 10] are *dataflow* languages that describe a graph of concurrent nodes. Adjacent nodes may communicate with one another along edges of that graph. This declarative paradigm is very amenable to static analysis, because the dataflow graph does not change at runtime.

By contrast, Esterel [11] and Céu [89]—as well as the SSM-based languages described in this dissertation—are *imperative* languages: programs prescribe statements executed by concurrent processes. Imperative programming emphasizes the transfer of *control flow*, and is widely used by many general-purpose programming languages like C, Lua, and Haskell.[10] One advantage of this paradigm is that it can encode program state using control flow. For instance, the program shown in Figure 1.3 uses imperative control state to encode a simple state machine that distinguishes single and double clicks. This expressiveness comes at the cost of analyzability, which needs to account for exponentially many combinations of process control state.

Other languages [69, 77] are based on the actor model of computation [1], which combines aspects of dataflow and imperative programming. *Actors* are locally stateful objects that communicate via message-passing. Externally, they behave like nodes of a dataflow graph, but internally, they may execute imperative statements encapsulated from their declarative environment.

[10] Although Haskell is a pure, functional language, its *monads* [97] can be used to build routines of effectful, imperative statements. Haskell provides syntactic sugar—**do**-notation—which programs may use to compose sequences of monads in a way that visually resembles the statements of a traditional imperative language.

```
present S else
    emit T
end
||
present T else
    emit S
end
```

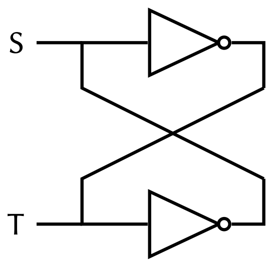


Figure 1.4: A nondeterministic Esterel program, left, that is logically equivalent to the bistable circuit diagram shown on the right. It is nondeterministic because there are two possible behaviors: one where S is present and T is absent, and another where S is absent and T is present.

1.6 Deterministic Concurrency

In many cases,[11] a useful programming model ensures the system is *deterministic*, which means that given the same inputs, the system has exactly one behavior [66]. Deterministic systems are easier to reason about—there is only one outcome to consider—and easier to test. Otherwise, in a nondeterministic system, successful test runs cannot conclusively rule out failures which are improbable but still possible.

Determinism is well-established in *sequential* settings, where there is only one thread of execution. Imperative languages use *sequential composition* (i.e., the "**;**" operator) to impose a total order on side effects; for purely functional lambda calculi, the Church-Rosser theorem guarantees the uniqueness of fully-reduced terms.[12] However, determinism often eludes *concurrent* systems, where there may be many ways to interleave the execution of multiple unsynchronized processes.

Alas, concurrency is a core aspect of real-time systems—physical time is essentially a concurrent process that advances alongside the program. To avoid the issues that come with nondeterminism, many languages designed for reactive real-time systems are based on programming models that support some form of *deterministic* concurrency.

[11] Not all use cases require determinism; some benefit from nondeterminism. For example, Signal [8, 10]—part of the same synchronous language family as Esterel and Lustre—does not enforce determinism because it was designed for *modeling* systems that are not necessarily deterministic. By contrast, my dissertation is focused on languages for *building* reactive real-time programs.

[12] In other words, normal forms are unique, and it does not matter what reduction strategy was taken to get there. Church and Rosser first stated and proved this theorem in 1936 [20]; Edwards provides a more accessible explanation and points out its relationship to determinism [30].

Synchronous and discrete-event languages already confer some degree of determinism "for free." They are unambiguous about whether one instant should evaluate before another, because instants are totally ordered according to timestamps or tags. However, nondeterminism can still manifest when logically simultaneous computations are not reconciled in a deterministic manner. For instance, in Verilog [55] and VHDL [56], zero-delay feedback loops lead to inconsistent behavior between simulations and synthesized hardware.

To enforce deterministic concurrency, some languages provide mechanisms to detect and reject nondeterminism at compile-time, while others use a restricted model of concurrency that is deterministic by construction. For instance, Esterel [11] rejects the program shown in Figure 1.4 because it describes two possible behaviors. Lingua Franca (LF) [70], Céu [89], and the SSM-based languages presented in this dissertation take the latter approach and use an evaluation strategy based on *priorities*. All parts of a system are evaluated at most once per instant, with simultaneous events handled according to priority. The system is deterministic as long as priorities are unique and totally ordered, because side effects (both internal and external) will happen in that order.

My dissertation's insistence on determinism alludes to the importance of human factors in the design and implementation of reactive languages, but my dissertation stops short of investigating these factors. Lee makes similar allusions in his article on deterministic models, where he frames this matter as one of "predictability" [65]: even though determinism ensures *repeatable* behavior, it does not mean that behavior is *predictable* for human reasoning. The work presented here aspires to be the basis of future work investigating these human factors.

1.7 Related Work

Many of the works described in this section are programming languages, but they are relevant to my work because of the programming models they embody. For the purposes of comparison, I conflate these concepts and use the terms "language" and "model" interchangeably; I refer to "SSM" as if it were a single language, SSLANG, even though implementations of this model exist in other languages.

Lingua Franca (LF) [70] is a discrete-events programming framework for reactive real-time systems. LF implements Lohstroh and Lee's reactor model of computation [69], which is inspired by (and a pun on) Hewitt's actor models [46]. The LF framework features a coordination language which configures a static dataflow graph of locally stateful "reactors." LF programs compile to one of several general-purpose target languages, including C, TypeScript, or Rust. The target language is also used to define the internal behavior of each reactor in an imperative style: statements from the target language are embedded within the LF coordination language, and quoted into the generated code.

Lohstroh and Lee developed LF around the same time Edwards and I worked on SSM, and our programming models share many similarities. Its execution model also uses two priority queues to schedule pending events and computation, avoiding the need to run the system at every instant. The key difference between SSM and LF is that LF demands and utilizes far more knowledge of the structure and behavior of its systems, trading flexibility for analyzability. LF fixes the topology of all reactors at compile time, statically determining reactor execution order based on explicit data dependencies. By contrast, SSM allows processes to spawn other processes at runtime, forming a dynamic process tree where execution order is determined by the priorities specified at parallel function call sites.

Ptides [99] is a discrete-event programming model for distributed, reactive systems, and underlies the PtidyOS real-time operating system [100]. Ptides was also developed by Lee's group at UC Berkeley and is a precursor to Lingua Franca's reactor model. Its actor graph is configured and simulated in the Ptolemy II design environment [33]; actors are C++ classes whose methods can directly perform system calls or access hardware. Ptides supports deadline-driven execution strategies that allow parts of the system to "time warp" local (logical) time ahead of global (physical) time [101]; Lingua Franca inherits this capability.

Timber [12] is a real-time systems language based on Nordlander's "reactive objects" [77]. Like actors, reactive objects are locally stateful and communicate with one another through timestamped messages. They are a collection of effectful, monadic computations called "reactions" that execute when a reactive object receives a message. To implement reactions, Timber features a pure, functional language based on (and roughly as expressive as) Haskell, which supports type classes, algebraic data types, and modules. However, unlike Haskell, Timber uses a strict evaluation strategy so that its execution time may be easier to predict.

Like SSM, Timber is deterministic, but the middle of its three "semantic layers" [17]—the reactive layer—is a nondeterministic process calculus that extends the (deterministic) functional bottom layer to a concurrent environment. Timber recovers determinism in its top layer, the scheduling layer, where asynchronous messages (internal events) are ordered by timestamp and deadline; simultaneous messages are handled in FIFO order.

Céu [88] is an imperative language based on discrete events and sparse timing, and was developed by Sant'anna and Ierusalimschy at PUC-Rio. Like SSM, Céu executes parallel branches according to their syntactic order, with earlier writes overwritten by subsequent writes in the same instant, though it discourages the use of non-commutative **par** branches. Unlike SSM, its **await** primitive supports blocking on both events and concrete durations (e.g., **await** 2ms). Internal events also work a little differently: **emit** e is a blocking operation that executes any process blocked on **await** e (in the order those **await** statements appear in the source code); after each such process blocks or terminates, computation resumes at the statement following **emit** e. Compared to SSM, Céu strikes a different balance in language design, trading expressive power for memory efficiency [89]. It foregoes recursive functions and heap allocation to ensure its programs use a constant amount of memory.

Esterel [11] is a synchronous language developed by French researchers Berry and Cosserat in the 1980s. Unlike its contemporaries, Lustre and Signal, Esterel is imperative: programs consist of (logically) parallel processes that communicate via signals. Esterel enforces a policy called signal coherence, which requires that each signal resolves to a single value at each instant. Under this policy, all readers of a signal may only execute after a writers, establishing causal order. Though formally sound, signal coherence precludes read-modify-write behavior, significantly restricting the imperative paradigm. This interpretation of causality is also difficult to explain and implement, because Esterel's signal graph is implicit [83]. Hanxleden et al.'s sequentially constructive concurrency [91, 94] relaxes signal coherence to allow signals with multiple values per instant, provided those values are totally ordered by explicit sequencing, but is even more complex. SSM uses a simpler scheme to schedule processes, whose relative priorities stay the same between instants.

Reactive C (RC) [13] was developed by Boussinot alongside his work on Esterel (with de Simone [14]) and features many analogous language constructs. RC is a synchronous language like Esterel, but favors a simpler implementation over mathematical rigor. In particular, it eschews compile-time causality analysis by enforcing signal coherence at runtime. RC is implemented as a preprocessor that emits C code; RC programs may embed arbitrary C expressions that evaluate as instantaneous statements, and are directly quoted into the generated code. However, the preprocessor is unaware of side effects, so the user must explicitly synchronize C expressions that need to be evaluated in a well-defined order.

SL [15] is an imperative synchronous language by Boussinot and de Simone whose syntax and semantics both closely follow Esterel's. Its main contribution is an adjustment to Esterel's **present** primitive, which tests for the presence or absence of a signal. In SL, reactions to signal absence (the **else** branch of a **present** statement) are delayed by one instant. This policy limits what parallel processes can instantaneously communicate, but it ensures programs are always causal and eliminates the need for complex causality analysis.

ReactiveML [71, 72] integrates Esterel processes and valued signals with a strongly-typed, functional language based on OCaml, bringing first-class functions and algebraic data types to the imperative synchronous paradigm. Developed by Mandel and Pouzet, ReactiveML uses a dynamic scheduler that supports spawning new processes at runtime; its programs are compiled to OCaml (which uses a heap). ReactiveML's model of computation is based on SL [15], which forbids immediate reactions to the absence of a signal.

Lustre [41, 42] is a synchronous language developed by French researchers Caspi and Halbwachs. Lustre was inspired by Wadge and Ashcroft's dataflow language Lucid [95]: its variables represent data "flows," which are infinite streams of values computed by the program. Lustre executes according to the ticks of a logical clock, but can construct slower clocks using the `when` keyword: `x when clk` produces a flow that is only valid during instants where the Boolean flow `clk` is **true**. Thus, Lustre can simulate aperiodic, event-driven systems as long as its base clock has sufficient resolution, but the system must run at each tick of the base clock to sample flows for the presence of an event.

Signal [8] is a synchronous specification language developed by French researchers Benveniste, Le Guernic, et al. around the same time as Lustre and Esterel. Unlike SSM (and most of the other languages discussed here), Signal is designed to be nondeterministic: programs describe sets of relations between signals, which may have more than one satisfying outcome. This allows Signal to model reactive systems that are not necessarily deterministic.

Giotto [45] is a dataflow-oriented language for periodic tasks, and measures logical time in concrete units (e.g., 3 ms) like SSM, LF, and Céu. Giotto tasks are "time-triggered," meaning they are scheduled according to fixed time intervals rather than in reaction to timed input events. Tasks (logically) execute for the entirety of their period but communicate with one another instantaneously; this is the opposite of how SSM works, where execution is logically instantaneously but communication (via delayed assignments) takes time. Tasks can also induce "mode switches" to replace the current set of running tasks with another. While this programming model is less flexible compared to reactive languages like SSM, it makes Giotto amenable to static analysis and scheduling.

Copilot [80] was designed by Pike et al. with NASA to build reliable runtime monitors. Copilot is a dataflow-oriented language that is implemented as a deeply embedded domain-specific language in Haskell [49]. It compiles to C like Scoria, our earliest SSM implementation (see Section 3.2), but uses a different programming model and embedding strategy. Copilot programs process "streams" of values, which are periodically sampled from the environment (what the application is monitoring). The syntax and semantics of Copilot streams are based on Haskell's lazy lists, though operations are restricted to those that work on infinite streams (e.g., Copilot programs are not allowed to ask for the "length" of a stream).

Functional reactive programming (FRP) [35] uses functional programming as the basis for describing reactive dataflow graphs. FRP was originally introduced by Elliott and Hudak for building interactive animations, and is the basis of many subsequent reactive programming frameworks [23, 25, 58, 75, 76]. Like SSM, FRP emphasizes high-level abstractions such as stream combinators and higher-order signals[13] at the expense of efficiency [62]. However, FRP signals are defined over a *continuous* model of time, in contrast to SSM and other synchronous models where time advances in *discrete* instants.

To improve the responsiveness of FRP programs, Elliott has proposed the inclusion of event-driven "push" semantics to complement FRP's traditionally sample-driven "pull" semantics [36]. In push-pull FRP, events are sparse and discrete like in SSM, but simultaneous events are handled differently: a signal may emit more than one event with the same timestamp, triggering multiple reactions in the same instant. SSM's scheduled variables can convey multiple instantaneous updates, but each process can only react once per instant.

[13] Higher-order signals are signals that carry other signals, and can simulate the behavior of dynamic dataflow networks that are otherwise impossible to express using a static dataflow graph.

Chapter 2: The Sparse Synchronous Programming Model

In reactive systems, concurrency and simultaneity are inevitable. Programs must interact with an environment that advances at its own pace, and independent interactions may coincide. Classical models of computation, based on sequential execution, sidestep this problem by nondeterministically interleaving concurrent execution (e.g., in C/C++), or by nondeterministically linearizing external events (e.g., in JavaScript). However, nondeterminism makes reactive systems difficult to develop, reason about, and test.

With the Sparse Synchronous Model (SSM) [31, 53], we sought to provide a basis for real-time reactive programming that treats deterministic concurrency and timing as a core concern. Our programming model is designed to extend conventional programming languages with precise, platform-speed-independent timing specification, and is the foundation of my thesis.

In this chapter, I describe our programming model, starting with an informal overview in Section 2.1. I provide a formal treatment of its semantics in Section 2.2,[1] followed by some discussion on our choices Section 2.3. SSM code examples are written using a Haskell-like syntax with with SSM-specific primitives like **wait** and **after** highlighted in red.

[1] Xijiao Li contributed insights toward the formulation of our reduction semantics.

```
x ← ref v            -- Allocate a scheduled variable with initial value v; bind it to x
v ← deref x          -- Read the current value of x; bind it to v
d ← since x          -- Read how long it has been since x was last assigned; bind the duration to d
assign x v           -- Assign value v to scheduled variable x
after d x v          -- Schedule a delayed assignment of v to x after some non-zero duration d
wait x               -- Block until an assignment to scheduled variable x
wait (x, y)          -- Block until an assignment to x, y, or both
```

Figure 2.1: Glossary of SSM primitives for scheduled variables.

```
1  sleep d = do
2      x ← ref ()
3      after d x ()
4      wait x
```

```
5  timeout d c = do
6      x ← ref ()
7      after d x ()
8      wait (x, c)
```

```
9  measureDuration e = do
10     timer ← ref ()
11     _ ← e
12     since timer
```

Figure 2.2: Implementation of sleep, timeout, and measureDuration in SSM; `()` denotes a constant unit value and is used here as an arbitrary value that we assign to scheduled variables.

2.1 Informal Semantics

2.1.1 Scheduled Variables

SSM uses *scheduled variables*[2] as its core synchronization and timing mechanism; Figure 2.1 lists its primitives for interacting with scheduled variables. Scheduled variables are mutable variables which store the value and logical time they were last assigned. They are so-called because they support *delayed assignments*, which schedule an assignment event to take place at the beginning of a later instant. Programs can also suspend execution until the next time a scheduled variable is assigned.

Figure 2.2 demonstrates some useful temporal abstractions we can implement using scheduled variables. The first function, `sleep`, appears in the `reactionTime` example from Chapter 1, and suspends execution for the given duration `d`. It uses a scheduled variable `x` as an alarm. With the **ref** primitive, `sleep` initializes `x` on line 2 to the unit value `()`. Then, on line 3, `sleep` schedules a delayed assignment to `x` using the **after** primitive, which will take place after the specified duration `d`. **after** itself evaluates instantaneously, so in line 4, `sleep` uses the **wait** primitive to

[2] In other work, we have referred to scheduled variables as references [53, 63], channel tables [52], or even just “variables” [31].

suspend execution until x is assigned due to the preceding **after** statement. Assignment events do not have to change the value of a scheduled variable: **wait** unblocks even though x is assigned the same unit value () it was initialized with.

The second function, timeout, is a variation on sleep that waits on a scheduled variable c with a timeout of duration d. Like sleep, it sets up an alarm using a scheduled variable in lines 6 and 7, but in line 8 it **wait**s on two scheduled variables at the same time. **wait** is *disjunctive*, meaning it unblocks as soon as one of its operands is assigned, so in this case, line 8 will unblock when either x or c are assigned, whichever happens earlier. SSM assumes it is possible for events to be logically simultaneous, meaning it is possible for x and c to be assigned in the same instant; in this case, **wait** will still unblock as expected.

The last function in Figure 2.2, measureDuration, is also featured in the reactionTime example. It measures the duration elapsed while evaluating the computation passed via e.[3] Before evaluating e, this function creates a scheduled variable which it uses as a logical timer. After evaluating e, it uses the **since** primitive to get the duration since timer was last assigned—when it was constructed in line 10. If e terminates instantaneously, then **since** will return a duration of 0.

Without **since**, **after**, and **wait**, scheduled variables have essentially the same interface and behavior as ML references or Haskell IORefs. The effects of instantaneous assignments are "felt" in the same instant, and are not overwritten by delayed assignments until a later instant. For instance, after executing the following sequence of statements:

```
assign x 10 ; assign x 20 ; after (secs 42) x 30 ; wait x
```

the current value of x is 20, the value it was last assigned (10 was overwritten). The value 30 is scheduled, but not assigned until 42 seconds later.

SSM measures durations in concrete units like milliseconds and minutes. They can be constructed with library-provided functions like msecs and mins and used as the first argument to

[3] The implementation of measureDuration shown here assumes that its argument is lazy, meaning the computation e is not evaluated until it is run in the **do** block, on line 11. A strict language can simulate this deferred evaluation by wrapping e in a closure and explicitly invoking that closure inside measureDuration, i.e., calling e **()** instead of just e.

```
1  sum r1 r2 r3 = do
2       par (wait r1, wait r2)
3       v1 ← deref r1
4       v2 ← deref r2
5       assign r3 (v1 + v2)

6  fib n r | n < 2       = after (secs 3) r 1
7          | otherwise = do
8              f1 ← ref 0
9              f2 ← ref 0
10             par (fib (n-1) f1,
11                  fib (n-2) f2,
12                  sum f1 f2 r)
```

Figure 2.3: A contrived Fibonacci example in SSM.

the **after** primitive. Our implementations of SSM use a fixed-precision representation for these duration values; the formal implications of this choice are discussed in Section 2.3.2.

SSM does not provide any mechanism to cancel a delayed assignment, nor does it specify how a scheduled variable should behave when a program attempts to schedule more than one delayed assignment to it. Scheduling multiple delayed assignments may be considered an error that is easily caught at runtime, though the runtime implementation I describe in Chapter 4 simply overwrites any previously scheduled delayed assignment.

2.1.2 Processes

SSM programs may execute multiple computations "at the same time" by spawning *processes* using the **par** primitive. Processes execute and suspend independently from each other, but they may use scheduled variables as synchronized communication channels. SSM allows programs to create any number of processes and scheduled variables, and does not assume any static dataflow paths. Instead, it ensures determinism by scheduling processes according to their *priority*.

The Fibonacci example in Figure 2.3 demonstrates how processes can pass data to one another using a mix of instantaneous and delayed assignments. On the left, `sum` is a helper function that propagates the sum of two scheduled variables when they are assigned. On line 2, `sum` **wait**s until `r1` and `r2` are assigned. It blocks on these scheduled variables in **par**allel. The **par** primitive spawns processes fork-join style and blocks until all its spawned processes have terminated, which is important for ensuring that line 2 blocks until *both* `r1` and `r2` have been assigned, in any order. In lines 3 and 4, `sum` **deref**erences `r1` and `r2` to read their newly assigned value, and (instantaneously)

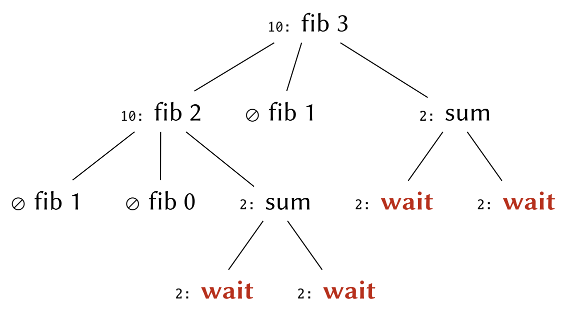


Figure 2.4: The process tree at the end of the initial instant where fib 3 is called. Each process is annotated with the line number being executed; ⊘ denotes that the process has terminated.

assigns their sum to r3.

On the right of Figure 2.3, fib uses sum to compute the nth Fibonacci number. The parameter n is given by value, alongside a scheduled variable r where the computed result is to be written. In the base case on line 6, when n is less than 2, fib schedules a delayed assignment **after** an arbitrarily chosen delay of 3 seconds. Meanwhile, in the recursive case from lines 7 to 13, fib allocates two scheduled variables f1 and f2 using the **ref** primitive, to hold the results of the two recursive calls. It performs those recursive calls as separate **par**allel processes, alongside a sum process that waits to sum the results once f1 and f2 are assigned.

Within an instant, SSM enforces deterministic concurrency by ensuring that processes are totally ordered according to their *priorities*. These priorities are determined by the syntactic order processes appear in SSM's **par** primitive. For instance, in the fib function, fib (n−1) f1 has the highest priority of the three, followed by fib (n−2) f2, followed by sum f1 f2 r; it is crucial that sum has a lower priority than the fib processes, to ensure that sum "sees" the instantaneous assignments from the recursive fib calls.

Most computation in SSM is instantaneous, aside from **wait** and **par** expressions. **wait** expressions suspend the running process, and block until the next instant where one of its arguments is assigned. Meanwhile, **par** blocks if and only if any of its spawned processes block. In other words, **par** is instantaneous if all child processes terminate without blocking. Notably, **after** expressions *do not block*: they instantaneously schedule computation for a future instant (they require a non-zero argument).

Thus, in the Fibonacci example in Figure 2.3, model time only advances on line 2 in `sum`, and on lines 10 to 13 in `fib`. Figure 2.4 illustrates the process tree formed by **par** expressions at the end of the instant where `fib 3` is called. Though the `fib 1` and `fib 0` processes have already terminated, the delayed assignments they scheduled will take place **after** 3 seconds, causing the **wait**ing processes to unblock. Since all computation after this initial instant is instantaneous (in particular, the **assign**ments from line 5), `fib n` will terminate after precisely 3 seconds for all $n \geq 2$ (`fib 0` and `fib 1` terminate instantly).

2.2 Formal semantics

This section gives a formal account of SSM by way of SSMΛ, a call-by-value lambda calculus that highlights the essence of our programming model. Its syntax is shown in Figure 2.5. SSMΛ is based on Pottier and Rémy's ML-the-calculus [84] and uses Felleisen-style contexts [38, 39] to enforce evaluation order. We define its small-step operation semantics with one set of rules for advancing between instants (Figure 2.7), and another for execution within an instant (Figure 2.8). Figure 2.9 illustrates how this term rewriting system operates on a small example.

2.2.1 Programs

SSMΛ programs are expressions e that consist of (curried) function applications, parallel expressions, and values v. The values include primitive functions that, when applied, correspond to primitives from the programming model introduced in Section 2.1. For example, **after** `d x v` becomes the expression **after** $d\ x\ v$, while **wait** `(x, y, z)` is written **wait** $x\ y\ z$. All values are expressions, but applications and parallel expressions are not values.

SSMΛ evaluates its arguments strictly,[4] so par cannot be a primitive function like **after** or **wait**; otherwise, **par** $f\ g$ would sequentially evaluate f and g rather than spawn processes to evaluate them in parallel. Instead, **par** is an associative prefix binary operator in SSMΛ: **par** `(a, b, c)`

[4] SSM does not require SSMΛ to be strict (call-by-value). In fact, the examples in Chapter 1 and Section 2.1 were deliberately vague about using a lazy language for the sake of a cleaner presentation. For background, see Plotkin [81] and Felleisen [37], who both explore the formal relationship between strict and lazy languages.

e ::=		**Expressions**
|	v	*Value*
|	$e\ e$	*Application*
|	**par** $e\ e$	*Parallel evaluation*
v ::=		**Values**
|	x	*Named variable*
|	d	*Duration value*
|	$()$	*Unit value*
|	m	*Memory location*†
|	$\lambda\, x\,.\, e$	*Function*
|	$\mathbf{ref}_1$	*Allocation*
|	$\mathbf{deref}_1$	*Dereference*
|	$\mathbf{since}_1$	*Time since last assignment*
|	$\mathbf{after}_3$	*Delayed assignment*
|	$\mathbf{assign}_2$	*Instant assignment*
|	$\mathbf{wait}_+$	*Wait for updates*
|	$\mathbf{check}_+$	*Check for updates*†
|	$\mathbf{block}_+$	*Suspend execution*†

s ::=		**Suspended programs**
|	**block** v^+	*Suspended primitive*
|	**par** $s\ s$	*Suspended parallel*
|	$\mathcal{E}[s]$	*Suspended evaluation*
|	v	*Completed evaluation*
$\mathcal{E}$ ::=		**Evaluation contexts**
|	$\bullet$	*Hole*
|	$\mathcal{E}\ e$	*Left of application*
|	$v\ \mathcal{E}$	*Right of application*
|	**par** $s\ \mathcal{E}$	*Left of parallel*
|	**par** $\mathcal{E}\ e$	*Right of parallel*

† Memory locations m, **check**, and **block** only appear in programs during evaluation.

Subscripts denote the arity of primitive functions like **ref** and **wait** (+ means one or more).

v^+ denotes one or more values.

Figure 2.5: Abstract syntax for SSMΛ. Programs are expressions e that may include values v. Suspended programs s cannot be reduced further in the current instant but have not terminated. Evaluation contexts $\mathcal{E}$ identify where reduction may occur within the context of a larger program; $\mathcal{E}[e]$ is the expression produced by substituting e into the hole $\bullet$ of evaluation context $\mathcal{E}$.

is equivalent to both **par** (**par** $a\ b$) c and **par** a (**par** $b\ c$). However, neither **par** nor its informal counterpart **par** are commutative: **par** $a\ b$ is not the same as **par** $b\ a$.

Three constructs are never produced directly from SSMΛ programs and are only created while an SSMΛ program is running: *memory locations*, used to index values stored in the heap; **check**; and **block**. These latter two encode **wait** expressions that are actively checking for updates or have suspended for the rest of the execution.

For brevity, we omit most features from realistic functional languages, but many can be incorporated in SSMΛ without affecting its core semantics. For instance, we can express let-expressions **let** $x = a$ **in** b using function application $(\lambda x\,.\,b)\ a$. Since SSMΛ is strict, sequenced statements $a\,;\,b$ also desugar to $(\lambda x\,.\,b)\ a$ with some fresh x that does not appear in b.

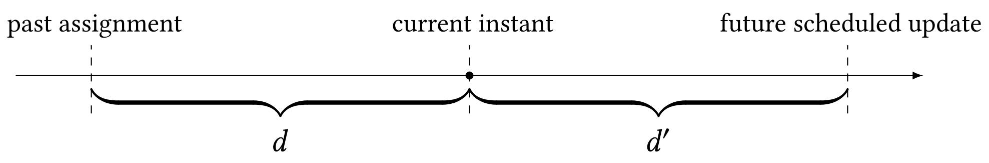


Figure 2.6: Illustration of what durations represent in the heap σ and the event queue ϕ, containing events $v \diamond d \in \sigma$ and $v' \diamond d' \in \phi$. Durations d and d' are relative to the current instant.

2.2.2 State

The state of an SSMΛ program includes a *memory heap* σ, and an *event queue* ϕ. Both are partial maps from memory locations m to events, which are *value-duration*[5] pairs written $v \diamond d$. We write $\sigma(m)$ and $\phi(m)$ to denote the event in σ and ϕ mapped to by m. To assign an event $v \diamond d$ to location m in the heap σ, we define

$$([m \mapsto v \diamond d]\sigma)(m') = \begin{cases} v \diamond d & \text{when } m' = m \\ \sigma(m') & \text{otherwise} \end{cases}$$

and the same for updating event queues $[m \mapsto v \diamond d]\phi$. Note that σ and ϕ are partial maps, so there can be at most one mapping from each memory location m, which means that

$$\sigma(m) = v \diamond d \ \text{ and } \ \sigma(m) = v' \diamond d' \quad \text{implies} \quad v = v' \ \text{ and } \ d = d'$$

and the same for ϕ.

Each memory location is uniquely associated with a heap-allocated scheduled variable and is used as an index in the heap and event queue, where that variable's metadata is stored. An event in the heap $v \diamond d$ represents the variable's "current" contents: v is its current value and d is a duration representing its age, i.e., how long since the variable was assigned v. Meanwhile, an event in the event queue $v' \diamond d'$ represents a pending update when the variable will be assigned v' after the duration d' elapses. Note that the durations in the heap and the event queue mean different things relative to the current instant, as illustrated in Figure 2.6.

[5] Note that SSMΛ's "events" do not use absolute timestamps (i.e., $v@t$ meaning "value v at time t"). The reasoning behind using durations is discussed in Section 2.3.1.

$$\text{S-R}{\scriptsize\text{EDUCE}}\quad \frac{\langle e\,,\,\sigma\,,\,\phi\rangle \rightarrow \langle e'\,,\,\sigma'\,,\,\phi'\rangle}{\langle \mathcal{E}[e]\,,\,\sigma\,,\,\phi\rangle \rightsquigarrow \langle \mathcal{E}[e']\,,\,\sigma'\,,\,\phi'\rangle}$$

$$\text{S-T}{\scriptsize\text{ICK}}\quad \frac{\begin{array}{c} d > 0 \qquad \nexists\, m',\ \phi(m') = v \diamond d' \text{ and } d > d' \\ \sigma'(m) = \begin{cases} v \diamond 0 & \text{when } \phi(m) = v \diamond d \\ v \diamond d' + d & \text{when } \sigma(m) = v \diamond d' \\ \textit{undefined} & \text{otherwise} \end{cases} \qquad \phi'(m) = \begin{cases} \textit{undefined} & \text{when } \phi(m) = v \diamond d \\ v \diamond d' - d & \text{when } \phi(m) = v \diamond d' \\ \textit{undefined} & \text{otherwise} \end{cases} \end{array}}{\langle s\,,\,\sigma\,,\,\phi\rangle \rightsquigarrow \langle\,[\mathbf{block} \rightarrow \mathbf{check}]s\,,\,\sigma'\,,\,\phi'\rangle}$$

Figure 2.7: Rules for the step relation $\rightsquigarrow$ between SSMΛ configurations ⟨*program, heap, event queue*⟩. S-Reduce takes a step within an instant (using reduction rules $\rightarrow$; see Figure 2.8); S-Tick advances time between instants on suspended programs, by duration d. For the definitions of σ' and ϕ' in S-Tick, cases above take precedence over those below when multiple conditions hold.

Events move from the queue to the heap in the S-Tick rule, when time advances. Programs can only schedule assignments to already-allocated variables, so any memory location present in the event queue must also be in the heap. However, the opposite is not always true: a memory location may be in the heap but not the queue when there is no pending update to that location.

2.2.3 Execution

The execution of an SSMΛ program proceeds in two alternating phases. In the first phase (S-Reduce), a program is reduced (evaluation contexts $\mathcal{E}$ control the reduction order) to either a value v, indicating the program has terminated, or a suspended program s that indicates what the program will do in a future instant (refer to Figure 2.5 for possible forms of v and s). In the second phase (S-Tick), time is advanced, assignments scheduled for that time are made, and every part of the program that was waiting on an update (**block**) is awakened to check whether its memory locations were assigned (**check**).

Figure 2.7 defines the rules for the two phases, which proceed as a series of step relations $\rightsquigarrow$ between *configurations*, which are 3-tuples $\langle e\,,\,\sigma\,,\,\phi\rangle$ that consist of the expression we are evaluating e; the current values of scheduled variables, stored as events in the heap σ; and scheduled

$$\textsc{R-Beta}\quad \frac{}{\langle (\lambda x \,.\, e)\ v\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle [x \to v]e\,,\ \sigma\,,\ \phi \rangle} \qquad \textsc{R-Ref}\quad \frac{m \notin \mathrm{dom}(\sigma)}{\langle \mathbf{ref}\ v\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle m,\ [m \mapsto v \diamond 0]\sigma,\ \phi \rangle}$$

$$\textsc{R-Deref}\quad \frac{\sigma(m) = v \diamond d}{\langle \mathbf{deref}\ m\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle v\,,\ \sigma\,,\ \phi \rangle} \qquad \textsc{R-Assign}\quad \frac{}{\langle \mathbf{assign}\ m\ v\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle (),\ [m \mapsto v \diamond 0]\sigma,\ \phi \rangle}$$

$$\textsc{R-Since}\quad \frac{\sigma(m) = v \diamond d}{\langle \mathbf{since}\ m\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle d\,,\ \sigma\,,\ \phi \rangle} \qquad \textsc{R-After}\quad \frac{d > 0}{\langle \mathbf{after}\ d\ m\ v\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle (),\ \sigma,\ [m \mapsto v \diamond d]\phi \rangle}$$

$$\textsc{R-Join}\quad \frac{}{\langle \mathbf{par}\ v_1\ v_2\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle (),\ \sigma\,,\ \phi \rangle} \qquad \textsc{R-Wait}\quad \frac{}{\langle \mathbf{wait}\ m_1 \ldots m_k\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle \mathbf{block}\ m_1 \ldots m_k\,,\ \sigma\,,\ \phi \rangle}$$

$$\textsc{R-Unblock}\quad \frac{\exists\, i \in \{1, \ldots, k\},\ \sigma(m_i) = v \diamond 0}{\langle \mathbf{check}\ m_1 \ldots m_k\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle (),\ \sigma\,,\ \phi \rangle} \qquad \textsc{R-Block}\quad \frac{\nexists\, i \in \{1, \ldots, k\},\ \sigma(m_i) = v \diamond 0}{\langle \mathbf{check}\ m_1 \ldots m_k\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle \mathbf{block}\ m_1 \ldots m_k\,,\ \sigma\,,\ \phi \rangle}$$

Figure 2.8: Reduction rules between instantaneous configurations. R-Ref, R-Assign, R-After, R-Deref, and R-Since create, write, schedule, and read events on the heap and event queue; R-Wait, R-Unblock, and R-Block handle blocking waits on heap values.

assignments of those variables, stored as events in the event queue ϕ. The S-Reduce rule takes steps in the first phase to evaluate expressions within an instant, without advancing time; the S-Tick rule steps between instants by adjusting each duration in the heap and event queue.

2.2.4 Steps within an instant: the S-Reduce rule

The S-Reduce runs the program in an instant by taking small steps of the form $\langle e\,,\ \sigma\,,\ \phi \rangle \longrightarrow \langle e'\,,\ \sigma'\,,\ \phi' \rangle$ where the expression e is a tiny part of the program being evaluated (the *redex*), and both the heap σ and the event queue ϕ may be updated. The current (logical) time, however, does not change during these small steps. This is Berry's synchrony hypothesis [11]: instructions do not advance time; only scheduled events do.

The S-Reduce rule enforces a total evaluation order through the *evaluation context* $\mathcal{E}$ defined in Figure 2.5.[6] This syntactic form specifies a unique "hole" $\bullet$ where a reduction may occur within the context of a larger program. For example, the context $\mathcal{E}$ e allows a function to be reduced before its argument is applied, whereas $v\ \mathcal{E}$ mandates that the function be reduced to a value before the argument is evaluated. Together, these impose a left-to-right evaluation order: the function must be reduced before its argument.[7] Similarly, **par** $\mathcal{E}\ e$ and **par** $s\ \mathcal{E}$ regulate the evaluation order between par branches, forcing the left branch of a par to be fully reduced to a suspended program (or a value) before reductions to the right branch may begin. We write $\mathcal{E}[e]$ to denote the expression produced by substituting redex e into the (unique) hole of context $\mathcal{E}$.

Figure 2.8 lists the reduction rules used by R-Reduce within an instant:

R-Beta is the standard beta-reduction rule, which insists that its argument is a fully-reduced value v, i.e., call-by-value [81]. The notation $[x \rightarrow v]e$ means to substitute v for all free occurrences of x in e.

R-Ref allocates a scheduled variable on the heap with fresh memory location m and instantly assigns it to value v. Here, "fresh" means a new memory location that is not currently in the domain of the map, i.e., $\sigma(m)$ is *undefined*. The event $v \diamond 0$ added to heap is assigned age 0 because it was last updated this instant.

R-Assign instantaneously assigns the value v to previously allocated memory location m. Just like R-Ref, the event is given age 0, so even if the value v is the same as it was before (e.g., assigning a unit scheduled variable ()), the previous age is overwritten.

R-After schedules an assignment event strictly in the future, i.e., value v will be assigned to m on the heap after some non-zero duration d. Note that this operation will overwrite any pending event on m and is the only reduction rule that modifies the event queue (though S-Tick updates the queue between instants).

[6] This style of small-step operation semantics using evaluation contexts is due to Felleisen and Friedman [38, 39], and also used by Pottier and Rémy for ML-the-calculus [84].

[7] Not to be confused with *call-by-value*—enforced by R-Beta—which requires an argument needs to be reduced to a value before it is *applied*.

R-Deref returns the current value of (heap) memory location m.

R-Since returns the time elapsed since (heap) memory location m was last written.

R-Join terminates a **par** expression when both of its branches have terminated (were reduced to values v as opposed to suspended programs s). This rule returns unit value $()$, discarding the values of the two branches.

R-Wait forces a **wait** expression to block when it is evaluated (even if one of the listed memory locations has just been written) by turning it into a **block**, one of the choices for a suspended program s.

R-Unblock terminates a **check** expression (a rewritten **block**; see below) when at least one of the scheduled variables (memory location m_i) it is waiting on has been written in the current instant.

R-Block is the opposite of R-Unblock: when none of the memory locations **check** is waiting on were written in the current instant, it turns back into a **block** that can be awakened in a later instant.

S-Reduce applies these rules to reduce a program in a particular instant into either a value v, which cannot be further reduced, or into a suspended program s, which expresses programs that cannot be reduced further in the current instant.

2.2.5 Steps between instants: the S-Tick rule

Once the program has been reduced as much as possible in the current instant (i.e., transformed into suspended program form as defined in Figure 2.5), the S-Tick rule does the three things: it advances time by duration d, performs assignments scheduled for the new instant, and "wakes up" all blocked processes in the program.

The choice of the time step d is what gives the Sparse Synchronous Model its name: instead of always advancing by a constant time step, SSMΛ allows the implementation to choose any non-zero duration that does not skip past a scheduled assignment event $\phi(m')$. An efficient implementation will usually choose to "sleep" as long as it can and only resume at the time of the next queued event, but the semantics are such that a system may wake up before the next event, at which point it would discover there is nothing to do and eventually suspend again. Note that the choice of d is what makes ϕ behave as a priority queue: the bound on d is the lowest duration in ϕ, the soonest scheduled assignment event in the event queue.

With the selected time step d, S-Tick forms the new heap σ' and event queue ϕ' by removing every queued event with pending duration d from ϕ and writing their values to the heap, overwriting the previous value in σ. Adding an event to the heap here is similar to R-Assign, but only S-Tick dequeues events. S-Tick also adds d to the age of other memory locations in the heap and subtracts d from pending events remaining in the queue, because past assignments are now further in the past and scheduled assignments are closer to the present.

Finally, S-Tick replaces each **block** in the suspended program e with a **check**. Those **block** expressions were either **wait** constructs that had just executed and blocked (R-Wait), or were **check** constructs that continued to block because none of their awaited memory locations had been written since they were last reduced (R-Block). This rewrite transforms the suspended program into an expression suitable for R-Reduce, so that waiting expressions can check the heap for updates (R-Unblock and R-Block).

Configuration: $\langle e\,,\,\sigma\,,\,\delta\rangle$	**Next Rule**
$\left\langle \begin{array}{l} \textbf{let}\ x = \underline{\textbf{ref}\ 10}\ \textbf{in} \\ \textbf{let}\ y = \textbf{ref}\ ()\ \textbf{in} \\ \textbf{par wait}\ x\ ;\ \textbf{assign}\ y\ () \\ \quad \textbf{after}\ 2_{ms}\ \ x\ 30\ ;\ \textbf{wait}\ y \end{array},\ \{\},\ \{\} \right\rangle$	R-Ref
$\left\langle \begin{array}{l} \underline{\textbf{let}\ x = m_x\ \textbf{in}} \\ \textbf{let}\ y = \textbf{ref}\ ()\ \textbf{in} \\ \textbf{par wait}\ x\ ;\ \textbf{assign}\ y\ () \\ \quad \textbf{after}\ 2_{ms}\ \ x\ 30\ ;\ \textbf{wait}\ y \end{array},\ \{m_x \mapsto 10 \diamond 0\},\ \{\} \right\rangle$	R-Beta
$\left\langle \begin{array}{l} \textbf{let}\ y = \underline{\textbf{ref}\ ()}\ \textbf{in} \\ \textbf{par wait}\ m_x\ ;\ \textbf{assign}\ y\ () \\ \quad \textbf{after}\ 2_{ms}\ \ m_x\ 30\ ;\ \textbf{wait}\ y \end{array},\ \{m_x \mapsto 10 \diamond 0\},\ \{\} \right\rangle$	R-Ref
$\left\langle \begin{array}{l} \underline{\textbf{let}\ y = m_y\ \textbf{in}} \\ \textbf{par wait}\ m_x\ ;\ \textbf{assign}\ y\ () \\ \quad \textbf{after}\ 2_{ms}\ \ m_x\ 30\ ;\ \textbf{wait}\ y \end{array},\ \left\{m_x \mapsto 10 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{\} \right\rangle$	R-Beta
$\left\langle \begin{array}{l} \textbf{par}\ \underline{\textbf{wait}\ m_x}\ ;\ \textbf{assign}\ m_y\ () \\ \quad \textbf{after}\ 2_{ms}\ \ m_x\ 30\ ;\ \textbf{wait}\ m_y \end{array},\ \left\{m_x \mapsto 10 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{\} \right\rangle$	R-Wait
$\left\langle \begin{array}{l} \textbf{par block}\ m_x\ ;\ \textbf{assign}\ m_y\ () \\ \quad \underline{\textbf{after}\ 2_{ms}\ \ m_x\ 30}\ ;\ \textbf{wait}\ m_y \end{array},\ \left\{m_x \mapsto 10 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{\} \right\rangle$	R-After
$\left\langle \begin{array}{l} \textbf{par block}\ m_x\ ;\ \textbf{assign}\ m_y\ () \\ \quad \underline{()\ ;\ \textbf{wait}\ m_y} \end{array},\ \left\{m_x \mapsto 10 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{m_x \mapsto 30 \diamond 2_{ms}\} \right\rangle$	R-Beta
$\left\langle \begin{array}{l} \textbf{par block}\ m_x\ ;\ \textbf{assign}\ m_y\ () \\ \quad \underline{\textbf{wait}\ m_y} \end{array},\ \left\{m_x \mapsto 10 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{m_x \mapsto 30 \diamond 2_{ms}\} \right\rangle$	R-Wait
$\left\langle \begin{array}{l} \textbf{par block}\ m_x\ ;\ \textbf{assign}\ m_y\ () \\ \quad \textbf{block}\ m_y \end{array},\ \left\{m_x \mapsto 10 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{m_x \mapsto 30 \diamond 2_{ms}\} \right\rangle$	S-Tick
$\left\langle \begin{array}{l} \textbf{par}\ \underline{\textbf{check}\ m_x}\ ;\ \textbf{assign}\ m_y\ () \\ \quad \textbf{check}\ m_y \end{array},\ \left\{m_x \mapsto 30 \diamond 0,\ m_y \mapsto () \diamond 2_{ms}\right\},\ \{\} \right\rangle$	R-Unblock
$\left\langle \begin{array}{l} \textbf{par}\ \underline{()\ ;\ \textbf{assign}\ m_y\ ()} \\ \quad \textbf{check}\ m_y \end{array},\ \left\{m_x \mapsto 30 \diamond 0,\ m_y \mapsto () \diamond 2_{ms}\right\},\ \{\} \right\rangle$	R-Beta
$\left\langle \begin{array}{l} \textbf{par}\ \underline{\textbf{assign}\ m_y\ ()} \\ \quad \textbf{check}\ m_y \end{array},\ \left\{m_x \mapsto 30 \diamond 0,\ m_y \mapsto () \diamond 2_{ms}\right\},\ \{\} \right\rangle$	R-Assign
$\left\langle \begin{array}{l} \textbf{par}\ () \\ \quad \underline{\textbf{check}\ m_y} \end{array},\ \left\{m_x \mapsto 30 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{\} \right\rangle$	R-Unblock
$\left\langle \begin{array}{l} \underline{\textbf{par}}\ () \\ \quad () \end{array},\ \left\{m_x \mapsto 30 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{\} \right\rangle$	R-Join
$\left\langle\ ()\ ,\ \left\{m_x \mapsto 30 \diamond 0,\ m_y \mapsto () \diamond 0\right\},\ \{\} \right\rangle$	*Terminated*

Figure 2.9: How SSMΛ's semantics operates on an example. Outside of S-Tick, each of the reduction rules (prefixed by R-) operates through the S-Reduce rule's evaluation context on the redex (underlined). For brevity, this example assumes the existence of a primitive value 2_{ms} representing 2 milliseconds; a realistic program would construct this value using the msecs function.

2.3 Discussion

2.3.1 Relative and Absolute Time

The programming model presented in this dissertation is based on *relative* time: the **since** primitive only reveals how long ago a scheduled variable was assigned relative to the current instant. In fact, the formal model presented in Section 2.2 does not involve any timestamps that identify one instant from another. When time advances, the durations in the heap and event queue are simply updated around the next point in time considered "now."

The presentation here contrasts how SSM was introduced in past publications, where each instant was tagged with an *absolute* timestamp [51, 52, 53].[8] Those publications featured a **written** primitive which returns the timestamp of the instant when a scheduled variable was last assigned, as well a **now** primitive that reveals the timestamp of the current instant. Those models could easily recover **since** x using those primitives, subtracting **written** x from **now**.

Though **written** and **now** are strictly more expressive than **since**,[9] they create a problematic global dependency on the current time t. This dependency appears as a parameter over reduction steps $\xrightarrow{t}$ [53] and requires us to pick a sensible t in order to evaluate any expression, even for functions like sleep whose behavior does not depend on when they are called. The program configuration can maintain the current time to ensure time travels forward, but there is no obvious choice for what the timestamp of the initial configuration should be.

The ambiguity of "time zero" is also annoying in practice. When we implemented SSM using **written** and **now**, we wanted to initialize global time according to the real world (e.g., Unix time). However, doing so would have caused our test suite to be nondeterministic, because its behavior would be dependent on when the test suite was run. We were thus forced to initialize logical time at a constant value 0 defining the **written** and **now** primitives in terms of this otherwise meaningless epoch.

[8] SSM was originally proposed without any primitives for measuring time [31]. Our first prototype, Scoria [63], included a **changed** primitive which only says whether a variable was updated in the current instant.

[9] *Expressive* in the sense that we cannot express **written** and **now** from **since** using macros alone [37].

In light of that experience, SSM loses very little from trading **written** and **now** for **since**. We can approximate the "time starts at 0" policy by constructing a scheduled variable during the initial instant to keep time, and implementing **now** and **written** in terms of that epoch:

```
-- Record first instant as epoch...
epoch ← ref ()

-- "Absolute" time since first instant
let now = since epoch
```

```
-- "Absolute" time when x was written
let written x = do
    xd ← since x
    ed ← since epoch
    return (xd - ed)
```

An implementation of **since**-flavored SSM can still store absolute timestamps under the hood, as long as they are not directly exposed to user programs. Doing so is much more efficient than storing relative durations because absolute timestamps are stable: the runtime does not need to adjust them between ticks. This is what our SSM runtime does: it stores the timestamp of the current instant, and the timestamp of when scheduled variable was last updated, and computes the return value of **since** by subtracting latter from the former. Chapter 4 describes our implementation in more detail.

2.3.2 Representation of Time

SSM's notion of logical time is based on physical durations like milliseconds and minutes. Many physical models insist that time is represented using real numbers, but the formal definition of SSM presented in Section 2.2 only requires that durations form a totally ordered set with addition and subtraction. This allows our implementations of SSM to use fixed-precision duration values whose precision can be chosen according to the platform's capabilities. For instance, the RP2040 SSM runtime described in Chapter 4 uses integers that represent 62.5 ns increments, because its timestamp peripherals run at 16 MHz (see Section 4.3). Our choice is based on practical considerations—floating-point arithmetic can be costly or unavailable on some microcontrollers—but also has important formal implications.

Unlike real numbers, fixed-precision numbers are not *dense*. Formally speaking, a dense representation means that for any pair of values $t_0 < t_1$, there always exists a distinct value t' in between them, i.e., $t_0 < t' < t_1$. Although dense numbers tend to match our intuitions about physical time, they are prone to "Zeno conditions" where a program generates an infinite number of events, asymptotically approaching but never advancing past some point in time [64]. For instance, the following program induces a Zeno condition by sleeping for $\sum_{k=0}^{\infty} \frac{d}{2^k}$:

```
zeno d = do
  sleep d
  zeno (d / 2)
```

This geometric series converges to $2d$, so logical time will never advance past this point. With SSM's non-dense, fixed-precision representation of time, d / 2 will eventually evaluate to 0 and cause sleep to crash (evaluating **after** 0 causes a runtime error).

However, using fixed-precision numbers also comes with downsides. One of our goals for designing SSM to be "sparse" was to prevent programs from using the smallest time increment d_ε to ask for the "*very* next instant" (á la Esterel's **pause**), because we wanted each platform to use the best available timing resolution without compromising portability. Even though SSM hides the underlying precision by requiring durations to be constructed using functions like msecs and mins, it is still possible to "search for" d_ε by repeatedly halving any duration, similar to the adversarial zeno program from before:

```
dε = search (msecs 42)      -- Start from an arbitrary duration, e.g., 42 ms
  where search d | d / 2 == 0 = d
                 | otherwise = search (d / 2)
```

A program can advance to the very next instant using sleep d_ε, subverting our goal for sparse time. Another downside is that performing integer arithmetic—especially integer division—on fixed-precision numbers can lead to unexpected rounding errors (though floating-precision representations are also prone to rounding errors).

SSMΛ sidesteps this issue because it does not include any constructs to perform arithmetic on durations, though a realistic implementation of SSM should include safeguards against unchecked rounding errors. For instance, a safe division operator would return both the quotient and the remainder; a linter can ensure both values are checked.

```
wait2 a b = do
  wait (a, b)                 -- Suspend until at least one is assigned
  ta ← since a                -- Check when a was last assigned
  tb ← since b                -- Check when b was last assigned
  if ta == 0 then
    if tb == 0 then
      return ()               -- Both a and b were assigned just now: return immediately
    else wait b               -- Only a was assigned just now: only block on b
  else wait a                 -- Only b was assigned just now: only block on a
```

Figure 2.10: Using SSM's disjunctive **wait** primitive to implement a conjunctive wait2 function. We can use the same pattern to implement an *n*-way waitN function, but the number of cases grows exponentially.

2.3.3 Disjunctive and Conjunctive Waiting

Unlike synchronous languages like Esterel [11] and SL [15], SSM only allows processes to react to the *presence* of assignment events (and not their absence). Its **wait** primitive supports blocking on multiple scheduled variables in a *disjunctive* manner: **wait** (a, b) unblocks the instant *either* a or b are updated. We designed **wait** this way because it can be used to implement *conjunctive* waiting—unblocking when *both* a and b are updated; Figure 2.10 shows a function that suspends until two variables have been updated. The opposite is not true: a conjunctive **wait** cannot implement disjunctive waiting. The intuition behind this asymmetry is that processes can only influence scheduling decisions when they are awake. A woken process can choose to suspend itself again if it needs to sleep for longer, but a suspended process cannot spontaneously wake itself.

The approach taken by Figure 2.10 to implement conjunctive waiting does not scale well beyond two scheduled variables, because it requires an exponential number of cases to account for every possible order that each variable is updated. It also awakens the waiting process unnecessarily, to check for updates, so it is not very efficient. Instead, we can implement conjunctive waiting much more succinctly using the **par** primitive, which spawns multiple processes and blocks until all of them terminate:

```
par (wait a, wait b, wait c)
```

This pattern is used in Figure 2.3, on line 2.

```
par (foo,
       bar)
baz
```

(a) **par** runs foo at a higher priority than bar, but blocks until both foo and bar terminate. Thus, baz cannot run in parallel with foo and bar without being "pushed into" a nested (or 3-way) **par** expression.

```
spawn foo
spawn bar
baz
```

(b) **spawn** always resumes in the same instant after the new process suspends (or terminates), so foo and bar do not block baz from running in parallel. At each instant they are scheduled in sequential order (foo, bar, baz).

```
defer foo
defer bar
baz
```

(c) **defer** is non-blocking, so it does not run foo or bar until baz suspends (or terminates) in the current instant. All three processes run in parallel, but are scheduled in reverse order at each instant (baz, bar, foo).

Figure 2.11: Comparing three primitives for concurrency: **par**, **spawn**, and **defer**.

However, the **par**-**wait** pattern also has limitations. It only works when **wait** statements are embedded within an outer **par** statement, so it can be awkward to express statements with an outer disjunction such as "wait until all buttons (e.g., a, b, and c) have been pressed, or until we have timed out (e.g., on d)," which looks like:

```
par (wait (a, d), wait (b, d), wait (c, d))
```

The outer disjunction needs to be distributed into the inner conjunction, causing d to be repeated many times. Instead, a more expressive **wait** primitive may allow SSM programs to specify their intention more directly, such as:

```
wait (a | b | c) & d
```

where the "|" connective means "either" and the "&" means "both." We considered extending **wait** to support any combination of these connectives, but ultimately decided that SSM is simpler to understand and implement without it. With ssm-lua, we strike a compromise that is more expressive: its `wait` function supports a limited form of conjunctive waiting to compensate for the lack of a primitive **par** statement: (see Section 3.3.2).

2.3.4 Parallelism

The **par** primitive appears in most incarnations of SSM, but there are other ways to structure parallelism. ssm-lua, an embedding of SSM in Lua that we describe in Section 3.3, exposes a

```
makeLED num = do                          handleLED num led = forever do
    led ← ref Off                             value ← deref led
    extInitLED num                            case value of
    defer (handleLED num led)                     Off → extWriteLED num 1
    return led                                    On  → extWriteLED num 0
                                              wait led
main = do
    button ← makeButton BTN1     -- Spawns a high-priority input handler process
    led ← makeLED LED0           -- Defers a low-priority output handler process
    reactionTime button led (msecs 4200)
```

Figure 2.12: Implementing the handler pattern with **spawn** and **defer**. The makeLED function uses **defer** to create low-level output handler process forwards assignments to led to the LED hardware; the definition of makeButton is not shown but mirrors makeLED. Here, helper functions beginning with "ext" are used to initialize and control the active-low LED hardware (setting 0 turns it On).

```
makeLED num k = do                      main = do
    led ← ref Off                           makeButton BTN1 (λ button → do
    extInitLED num                              makeLED LED0 (λ led → do
    par (handleLED num led, k led)                  reactionTime button led (msecs 4200)))
```

Figure 2.13: Implementing the handler pattern using **par** is quite awkward compared to using **spawn** and **defer** (Figure 2.12). In order to run anything alongside the handler, the main function needs to pass a continuation k to makeLED, leading to "callback hell."

different interface based on **spawn** and **defer**. Both functions create a new process executing the given closure, at the next highest (**spawn**) or lowest (**defer**) priority relative to the calling process. Figure 2.11 shows how **spawn** and **defer** work, and highlights the key difference from **par**: **spawn** and **defer** allow the current process to resume execution in the same instant,[10] whereas **par** does not (unless all child processes terminate instantly).

The **spawn** and **defer** primitives offer more flexibility because they do not impose **par**'s fork-join structure. For instance, libraries can use them to construct "handler" processes that manage and monitor scheduled variables. This pattern is very useful for hardware peripherals like push buttons and LEDs, which appear to SSM programs as scheduled variables; Figure 2.12 shows an application of this use case. A peripheral's handler can encapsulate its configuration and encoding, while the application focuses on how those peripherals are used.

[10] **spawn** temporarily yields to the newly created process, but when it suspends (or terminates), control returns to where **spawn** was called; **defer** does not yield control until current process suspends (or terminates).

```
par (f, g) = do                       join fd gd = do
    fd ← ref False                        fs ← deref fd
    gd ← ref False                        gs ← deref gd
    spawn (f ≫ assign fd True)            if fs && gs
    spawn (g ≫ assign gd True)                then return ()
    join fd gd                                else wait (fd, fd) ≫ join fd gd
```

Figure 2.14: Implementing **par** using **spawn** and **wait**. The "≫" operator is borrowed from Haskell and used to sequence computations; f ≫ **assign** fd **True** means "run f, then **assign** fd **True**."

The handler pattern is quite awkward without **spawn** and **defer**, because **par** does not allow a library function to return control to its caller until all child processes have terminated. Figure 2.13 shows us that we can work around this limitation by passing a callback that is run alongside the handler, at the expense of increased indentation and obfuscated control flow.

Figure 2.14 shows how we can recover **par** using **spawn** and join, a helper function that emulates **par**'s blocking behavior. Although **spawn** and **defer** are more expressive than **par**, they require processes to dynamically redistribute their priorities, to avoid running out of priority numbers. Efficient algorithms for doing so exist [6, 29], but they are difficult to implement and were not included in the SSM runtime implementation described in Chapter 4.[11]

[11] ssm-lua is independent of this runtime and includes an implementation of Dietz & Sleator's tag-range relabeling algorithm; see Section 3.3.3.

Chapter 3: Sparse Synchronous Programming Languages

SSMΛ establishes a formal foundation for the Sparse Synchronous Model, but lacks too many language features to be suitable for programming. In this chapter, I present three different SSM language implementations that allow users to write programs based on our model.

The first is SSLang, a standalone functional language we designed as the "canonical" implementation of our programming model. It took many years to develop and was first introduced in our MEMOCODE 2023 paper [51], and evolved out of our FDL 2020 paper's "toy language" [31] and our TECS journal paper's proposed "SSML" [53].[1] SSLang uses the set of SSM primitives in Figure 2.1, but add many features such as **let**-polymorphism, algebraic data types, and pattern matching. It compiles to portable C code that links against the SSM runtime I describe in Chapter 4, and is used for the experiments I discuss in Chapter 5.

The next language, Scoria, is embedded in the Haskell programming language [63].[2] We built Scoria early on in the development of SSM to prototype our programming model and test its runtime. Scoria also compiles to C, but can also be interpreted by an implementation of SSM within Haskell. We compared their execution traces to validate our runtime implementation.

Finally, ssm-lua is implementation of SSM in the Lua programming language. We developed ssm-lua to explore the feasibility of exposing our programming model as a pure Lua library, without relying on our C runtime. ssm-lua relies on Lua's garbage collector and support for coroutines, adapts SSM's primitives to suit Lua's idioms and native data structures.

I use red to highlight SSM runtime-defined identifiers in the C code generated from SSLang; SSM-specific primitives in Scoria and ssm-lua are also highlighted in red.

[1] Significant contributors to SSLang not credited as paper authors include: Hans J. Montero, Xijiao Li, Feitong Qiao, Emily Sillars, Yiming Fang, Chris Yoon, Eric Fang, Anjali Smith, Hao Zhou, Elaine A. Ramos, and Wei Qiang. The name "SSLang" is due to Hans J. Montero.

[2] Most of the credit for Scoria's design and development goes to Robert Krook, with help from our other collaborators at Chalmers University, Bo Joel Svensson and Koen Claessen.

```
wait2 (x: &a) (y: &b) =
    par wait x
        wait y


sum (x: &Int) (y: &Int) (z: &Int) =
    wait2 x y
    z := deref x + deref y
```

```
fib (n: Int) (r: &Int) =
    if n < 2
        after secs 3, r := 1
    else
        let (f1, f2) = (ref 0, ref 0)
        par fib (n-1) f1
            fib (n-2) f2
            sum f1 f2 r
```

Figure 3.1: A contrived Fibonacci example based on Figure 2.3, written in SSLang.

3.1 SSLang: a Standalone Language for SSM

SSLang ("Sparse Synchronous Language") is a standalone programming language that implements the model introduced in Chapter 2. It compiles to portable C code that links against our SSM runtime to run on microcontrollers.

Figure 3.1 shows the Fibonacci example from Chapter 2 rewritten in SSLang; Figure 3.2 shows a small SSLang function that illustrates more of its syntax. Like SSMΛ, SSLang is a call-by-value language that evaluates its arguments strictly and relies on side effects to manipulate scheduled variables (see Section 2.2). SSLang's syntax bears some resemblance to the pseudocode used in Chapter 1 and Section 2.1, but there are some important differences due to SSLang not being based on Haskell's monadic **do**-notation. In SSLang, most newlines delimit **let**-bindings rather than monadic computations. The following are equivalent:

```
f a b               let _ = f a b;
let g x = y         let g x = y;
z                   z
```

The statements on the left desugars to the chain of **let**-expressions on the right, where the value of f a b is bound to the empty variable _ and thrown away (though side effects from evaluating f a b remain and are "felt" in the following lines). The inserted semicolons ";" have the same meaning as the "**in**" keyword from ML-family languages[3] and delimit the bound expression (after "=") from the body of a **let**-expression. In SSLang, ":=" replaces the **assign** primitive,[4] and also

[3] OCaml and Standard ML are prominent members of this language family.

[4] A white lie: previous publications [51, 53]—as well as the actual implementation of SSLang—use "<-" instead of the ":=" operator. I use the ":=" in this dissertation to avoid overlap with Haskell's "←"-notation.

```
foo (arg: &Int) -> Int =         // Define a function named foo with argument arg
    let loc = ref 10             // Create a new scheduled variable named loc, initialized to 10
    after msecs 400, loc := 20   // Schedule a delayed assignment to loc after 400 ms
    wait arg || loc              // Wait for an assignment to arg or loc
    loc := 30                    // Assign the value 30 to loc
    par bar loc || baz arg       // Call bar and baz in parallel, passing loc and arg
    deref arg                    // Return the current value of arg
```

Figure 3.2: A SSLang function which we compile to C. This example features type annotations, scheduled variables, and parallel function calls. Line comments begin with //.

appears in **after**-expressions, which are given their own keyword and syntax; "**par**" and "**wait**" are also keywords and precede one or more " || "-delimited expressions. Figure 3.1 demonstrates that the operands of a **par** expression can also be delimited using newlines to avoid ambiguity with **wait**, like in wait2, or simply for clarity, like in fib. The unary type constructor "&" in foo denotes scheduled variables; "&Int" means "a scheduled variable that holds Ints."

SSLang is a functional programming language with closures and higher-order functions. It borrows many language features from the ML family, including **let**-polymorphism [26, 84], user-defined algebraic data types [16], and pattern matching expressions [3, 96]. SSLang supports type inference, so type annotations such as those seen in Figure 3.2 are optional (though beneficial as documentation). When given, annotations are checked against inferred types and can also be used to specialize polymorphic terms (e.g., constrain a term of type a -> a to be Int -> Int). For automatic memory management, SSLang uses a reference counting garbage collector [22].

The central challenge of translating SSLang comes from supporting both recursion and concurrent processes. Typically, a simple call stack suffices for recursion, while stackless coroutines can be used to implement processes. However, as their names might suggest, call stacks and stackless coroutines are not directly compatible, because stackless coroutines prohibit deeply nested (possibly recursive) functions from suspending. Instead, SSLang uses *stackful coroutines* that allocate activation records on the heap.

To focus our discussion on sparse synchronous programming, this dissertation will omit the implementation details of these features; the rest of this section will discuss how we compile SSLang to C, using Figure 3.2 as a running example.

```
typedef struct {
    act_t      act;        // Common activation record header
    value_t    arg;        // Function argument
    value_t   *ret;        // Where to write return value
    value_t    loc;        // Local variable
    trigger_t trig_1;      // Used to wake up foo when it is waiting
    trigger_t trig_2;      // Used to wake up foo when it is waiting
} act_foo_t;

act_t *enter_foo(act_t *caller, uint32_t priority, uint8_t depth,
                 value_t *argv, value_t *ret)
{
    act_foo_t *act =
        (act_foo_t *) act_alloc(sizeof(act_foo_t), step_foo,
                                caller, priority, depth);
    act->arg = argv[0];
    act->ret = ret;
    act->trig_1.act = &act->act;
    act->trig_2.act = &act->act;
    return &act->act;
}
```

Figure 3.3: Activation record and enter function for foo from Figure 3.2, generated based on foo's type signature. The activation record structure act_foo_t starts with the common header from Figure 3.5.

3.1.1 Functions and Activation Records

Each SSLang function foo is compiled into two C functions via a syntax-directed translation: an *enter function* that allocates and initializes foo's activation record (Figure 3.3), and a *step function* that performs the work of foo in a single instant, e.g., from when it is resumed by some event to when it suspends (Figure 3.4).

Unlike C functions, SSLang functions have the ability to suspend and resume between the execution of other functions. To support this form of non-local control flow, the SSLang treats each function invocation as an individually schedulable process whose local state is maintained in a heap-allocated *activation record*. Each SSLang function has its own specialized activation record type that stores local variables, arguments, and other runtime data; Figure 3.3 shows the activation record layout for the foo function from Figure 3.2.

Each activation record starts with the generic act_t header shown in Figure 3.5. They are are laid out like this so that it is always safe to cast a pointer to a function-specific activation record to an act_t pointer. This header maintains information used to resume executing its suspended step

```
void step_foo(act_t *header)
{
    act_foo_t *act = (act_foo_t *) header;
    value_t ret;    // Where the return value will be temporarily assigned
    switch (act->act.pc) {
      case 0:

2: let loc = ref 10
        act->loc = new_sv(marshal(10));

3: after msecs 400, loc := 20
        later(act->loc, now() + msecs(400), marshal(20));

4: wait arg || loc
        sensitize(act->arg, &act->trig_1);
        sensitize(act->loc, &act->trig_2);
        act->act.pc = 1;
        return;
      case 1:
        desensitize(&act->trig_1);
        desensitize(&act->trig_2);

5: loc := 30
        assign(act->loc, act->act.priority, marshal(30))

6: par bar loc || baz arg
        uint8_t depth = act->act.depth - 1;
        uint32_t prio = act->act.priority;
        {   value_t argv[1] = { act->loc };
            activate(enter_bar(&act->act, prio, depth, argv, NULL));
        }
        prio += 1 << depth;
        {   value_t argv[1] = { act->arg };
            activate(enter_baz(&act->act, prio, depth, argv, NULL));
        }
        actg->pc = 2;
        return;
      case 2:

7: deref arg
        ret = deref(act->arg);
      default:
        break;
    }
    if (act->ret)
        *act->ret = ret;
    leave(&act->act, sizeof(act_foo_t));
}
```

Figure 3.4: Step function for foo from Figure 3.2. The source for each line of this syntax-directed translation is annotated in gray; temporary values are elided and memory management is omitted, for clarity.

```
typedef struct {                    // Generic activation record header
    void (*step)(act_t *);          // Step function, evaluates an instant
    act_t     *caller;              // Parent's activation record, i.e., return address
    uint16_t  pc;                   // Saved control state, an encoded program counter
    uint16_t  children;             // Number of children, if suspended at par
    uint32_t  priority;             // Used to determine order in the run queue
    uint8_t   depth;                // LSB of our priority
    bool      scheduled;            // Whether this process is in the run queue
} act_t;
```

Figure 3.5: Generic activation record header used by the SSM runtime.

function: a pointer to that step function, and its control state (an encoded program counter). The header also contains a pointer to the parent's activation record as well as the number of running children, so that the last child to terminate can resume its parent. Finally, the header holds some data used to make scheduling decisions: two numbers related to its scheduling priority, and a flag indicating whether the function has already been scheduled to run in the current instant.

Each SSLang function needs a dedicated enter function that knows the size and layout of its activation record: foo's enter function, shown in Figure 3.3, is generated by the SSLang compiler according to foo's function signature and activation record `act_foo_t`. The generic `act_alloc` helper allocates an activation record from the heap[5] and initializes its `act_t` header's fields, but leaves `enter_foo` to initialize foo-specific fields. When foo terminates, this activation record is freed by foo's step function (Figure 3.4) using the `leave` helper function; `leave` also decrements the record's `children` counter and reschedule's foo's `parent` (its caller) if `foo` was the last child to leave.

3.1.2 Representing Values

The `value_t` type used in foo's activation record to store arguments and local variables (Figure 3.3) is our runtime representation for all SSLang values. We use this uniformly-sized representation so that generated polymorphic code and the rest of the runtime system can handle these values without having treat types differently.

[5] Others have shown that storing activation records on the heap can give similar performance to stack allocation [2], but makes suspending and resuming processes much simpler to implement.

```c
typedef union {
  uint32_t  packed;          // LSB=1: 31-bit integer
  mm_t     *boxed;           // LSB=0: pointer to heap object header
} value_t;

#define on_heap(v)          (((v).packed & 0x1) == 0)
#define marshal(v)          (value_t) { .packed = ((v) << 1 | 1) }
#define unmarshal(v)        ((v).packed >> 1)

typedef uint64_t time_t;    // 64-bit timestamps are stored on the heap
```

Figure 3.6: Memory layout for the untyped runtime representation of SSLang values, and macros for testing, constructing, and deconstructing packed values. Also, logical timestamps are 64-bit integers.

Figure 3.6 shows the machine-word representation we use for values. For 32-bit processors (our main target), the bits represent either a 31-bit packed value or a pointer to a larger object on the heap. We restrict the packed values to 31 bits for code portability, even on processors with 64-bit pointers.

The least significant bit (LSB) of the machine word described by `value_t` distinguishes pointers from packed values: heap pointers are always word-aligned, so their LSB will always be 0; To distinguish them from pointers, packed values always have an LSB of 1; their remaining bits can be used to store types that require fewer than 31 bits of space. This "bit-stealing" technique is typical in functional programming languages; we based our implementation on OCaml [68]. Figure 3.6 shows macros that test whether a value is a pointer to a heap object, and for converting between normal C values and packed SSLang values.

SSLang stores and manages several kinds of objects on the heap, including scheduled variables, closures, arrays, binary blobs. Similar to activation records, each object starts with a generic `mm_t` header that stores information about their size, layout, and reference count. Its memory management system uses the layout to recursively decrement reference counts on outgoing pointers when an object is freed.

SSLang uses 31-bit integers and stores them as packed values, but uses heap-allocated 64-bit integers for its logical timestamps. We chose to use larger timestamps despite their overhead to avoid headaches that arise from wraparound: with nanosecond precision, 64-bit wraparound

```
typedef struct {                    // Scheduled variables
    mm_t        header;             // Memory management header
    value_t     value;              // Current (i.e., last assigned) value
    time_t      last_written;       // Time of last assignment
    value_t     later_value;        // Value of scheduled assignment
    time_t      later_time;         // Time of scheduled assignment
    trigger_t *triggers;            // Linked list of waiting processes
} sv_t;

void assign(value_t ref, uint32_t prio, value_t val);
void later(value_t ref, time_t time, value_t val);
```

Figure 3.7: Runtime representation for scheduled variables, and helper functions for instantaneous and delayed assignment.

only occurs once every 584 years;[6] 32 bits afford us less than 5 seconds. Even with microsecond precision, 32-bit timestamps wrap around in a little over an hour.

In general, algebraic data types are allocated on the heap, but SSLang tries to encode them as packed values where possible, so that they can be stored "inline" without using the heap. For example, the Booleans in SSLang's standard library are defined as:

type Bool = False | True

False and True are compiled to `marshal(0)` and `marshal(1)` and stored as packed values.

3.1.3 Scheduled Variables

SSLang's scheduled variables behave like mutable references in ML-family languages and hold the value most recently assigned to them. They are allocated on the heap according to the structure defined in Figure 3.7. In addition to its current `value`, scheduled variables also record when they were `last_written`. SSLang's **since** primitive uses this timestamp to compute how long it has been since a variable was last assigned by subtracting it from `now`.

Each scheduled variable may have at most one pending assignment event, whose update time and value are recorded in the `later_time` and `later_value` fields; when no event is pending, `later_time` is `ULONG_MAX` and `later_value` is undefined. When `later_time` arrives, the

[6] Comfortably outlasting the span of any author's research career.

```
typedef struct {                     // Node in process trigger list
    act_t       *act;                // Sensitive (waiting) process
    trigger_t   *next;               // Next process trigger node
    trigger_t  **prev_ptr;           // Back pointer (used for deletion)
} trigger_t;

void sensitize(value_t ref, trigger_t *trigger);
void desensitize(trigger_t *trigger);
```

Figure 3.8: Linked list of triggers, for waking up **wait**ing processes when a scheduled variable is written.

`later_value` is copied to `value` and any processes in the `triggers` list are resumed. Processes add themselves to that list when they **wait** on a scheduled variable.

Figure 3.7 also includes the type signature of some functions used to update scheduled variables. The `assign` function implements instantaneous assignment, and updates a scheduled variable's `value` and `last_written` fields. In addition to the value being assigned, `assign` also takes the priority of the current routine; it only schedules sensitive routines with a lower priority. The `later` function implements delayed assignment: it saves the future value and time to `later_value` and `later_time` and asks the runtime scheduler to queue the pending update event at `later_time`. If `later` is called on a variable that already has a pending event, the existing event is overwritten to avoid an unbounded accumulation of events.[7]

3.1.4 Suspending and Resuming

A process may suspend for one of two reasons: it is blocking on an assignment to a scheduled variable (**wait**), or it is blocking on called child processes to return (a function call or **par**). In Figure 3.4, foo's step function demonstrates both scenarios. At each suspension point, the step function updates the activation record's program counter `pc` with a "return address" and returns from the step function. The next time the step function is invoked, the switch statement resumes execution at the `case` immediately following the `return`, corresponding to the saved program counter.

Before suspending, the step function conveys a wake condition to the scheduler so that it does

[7] This behavior is not strictly required by SSM's semantics; another way to handle extraneous scheduled assignments is to throw a runtime error.

```
void leave(act_t *act, size_t size)
{
    act_t *parent = act->parent;
    act_free(act, size);                    // Deallocate the full activation record
    if (--parent->children == 0)
        parent->step(parent);               // The last child resumes the parent
}
```

Figure 3.9: Implementation of `leave`, which deallocates an activation record and may return to its parent.

not remain asleep forever. When a process suspends due to a **wait**, it adds itself to the trigger list of each scheduled variable it waits on. Triggers are used as a kind of condition variable [47]: when a scheduled variable is written, the runtime traverses through its trigger list and schedules any sensitive processes to execute in that instant. The trigger list is doubly-linked for constant time deletion; its node type definition is shown in Figure 3.8. Each node contains a pointer back to the waiting function's activation record (initialized in the enter function, such as in Figure 3.3), and is enqueued and dequeued using `sensitize` and `desensitize`.

A process can reuse its triggers across different **wait** suspension points, but must use a unique trigger for each scheduled variable it waits on. As such, its activation record needs to contain at least as many triggers as the maximum number of variables it waits on at one time. The foo function from Figure 3.2 waits on two scheduled variables, arg and loc, so its activation record needs two triggers.

Processes also suspend when they spawn one or more child processes; they resume when all those child processes terminate. In Figure 3.3, we see foo's enter function adds its parent (`caller`) to its activation record, so that it can revisit its parent while leaving; `enter_alloc` also increments the parent's `children` count. In Figure 3.4, foo's step function terminates the process when control breaks out of the switch statement. As shown in Figure 3.9, after `leave` frees the given activation record, it decrements the `children` count of its parent. If `leave` was called by the last child, it resumes the parent's step function.

3.1.5 Function Calls and Priorities

SSLang programs use **par** to evaluate multiple functions in parallel, ordered according to their position in the **par** expression. Single function calls are treated as unary "parallel" calls.

Our runtime does not directly support evaluating non-call expressions in parallel, such as the parallel **wait** in Figure 3.1's wait2 function. Instead, we transform these expressions to parallel function calls using a source-to-source translation. We recursively replace non-call parallel expressions with calls to lifted top-level functions, where local variables that appear free in each expression are passed as arguments to the lifted function. For instance, the wait2 function from Figure 3.1 is translated to:

```
__wait2_par1 x = wait x          wait2 x y = par __wait2_par1 x
__wait2_par2 y = wait y                          __wait2_par2 y
```

At a **par** call site, the parent calls each child's enter function to allocate their activation records (see Figure 3.2), which are passed to `activate` to schedule those child processes for execution.

Deterministic concurrency was a key design goal for SSM (thus SSLang). We achieve it in part by mandating that at each instant, **par** operands further to the left must evaluate before operands to the right. In our formal semantics, this evaluation order was enforced using an evaluation context that always awakens and checks the first branch of a **par** before the second branch. These wake-ups are wasteful when most **wait** statements stay suspended in an instant. Instead, when a scheduled variable is written, we only schedule each process that is blocked waiting on that variable.

We force the scheduled processes to run in each instant in the order prescribed by semantics by assigning a unique priority number to each active process. The scheduler then runs processes in priority order. In the case of a single function call, the child simply inherits the priority of its parent, which is unambiguous because only one of them is ever running at once.

When multiple function calls are evaluated using **par**, we assign priority numbers in a hierarchical manner that subdivides the range of priority numbers allocated to the caller. Each process has a priority-depth pair (p, d) where $p \geq 2^d$, that indicates it owns priority numbers p through

$p + 2^d - 1$. When a process calls k children, it assigns pairs (p, d'), $(p + 2^{d'}, d')$, $(p + 2 \cdot 2^{d'}, d')$, $\ldots$, $(p + (k-1)2^{d'}, d')$, where $d' = d - \lceil \log_2 k \rceil$. The depth may also be interpreted as the index of the least significant bit in the priority.

For example, if a process has the pair $(16, 4)$, it owns priority numbers 16 through $16+16-1 = 31$ and calls four children, the children are given pairs $(16, 2)$, $(20, 2)$, $(24, 2)$, and $(28, 2)$. And if the $(24, 2)$ child in turn calls two children, they would be given pairs $(24, 1)$ and $(26, 1)$. In Figure 3.4, the `depth` and `priority` variables dynamically compute the new priority-depth pairs at the call site for bar and baz.

Our runtime system uses 32-bit unsigned integers (`uint32_t`) to represent priorities and 8-bit unsigned integers (`uint8_t`) to represent depths. This provides four billion unique priority numbers, although a pathological program could exhaust them. We later addressed this issue in ssm-lua (Section 3.3) using Dietz & Sleator's *tag-range relabeling* algorithm [6, 29].

3.1.6 Calling C Functions from SSLang

Our language is designed to run on microcontrollers that provide C libraries to configure and interact with hardware peripherals, so SSLang provides a basic interface to C for calling functions provided by these libraries. SSLang programs may use the "extern" keyword to declare a foreign function symbol, annotated with a SSLang type signature, e.g.:

```
extern extWriteLED : Int -> Int -> Int
```

SSLang translates this to the equivalent declaration in the generated C output:

```
extern value_t extWriteLED(value_t, value_t);
```

The C declaration has the same number of arguments as in SSLang, but all types are erased due to SSLang's uniform value representation.

SSLang considers foreign function calls logically instantaneous, and assumes that they terminate in the same instant. Although it is possible to pass a SSLang closure to C, the behavior of invoking that closure in C is undefined because the native (C) stack frame will not be saved when that closure suspends.

```
sum :: Ref Word32 -> Ref Word32 -> Ref Word32 -> SSM ()
sum x y z = routine do
    par [wait x, wait y]
    assign z (deref x + deref y)

fib :: Exp Word32 -> Ref Word32 -> SSM ()
fib n r = routine do
    if n <. 2
      then after (secs 3) r 1
      else do f1 <- ref 0
              f2 <- ref 0
              par [fib (n-1) f1, fib (n-2) f2, sum f1 f2 r]
```

Figure 3.10: A contrived Fibonacci example based on Figure 2.3, written in Scoria. Module imports, LANG pragmas (for GHC extensions), and other preamble have been omitted for clarity.

3.2 Scoria: a Deep Embedding of SSM in Haskell

Compilers are complex pieces of software and cannot be written overnight. We spent many months developing the SSLang compiler, with most of our time spent on "uninteresting" parts orthogonal to our programming model like the parser and type checker. It was a while before we could compile interesting SSLang programs.

We had built our SSM runtime long before SSLang; to test it without a working compiler, we implemented a prototype language, Scoria [63], that also compiles into C code. Scoria is embedded in a host language, Haskell, and uses Haskell's parser and type checker as its own. Scoria is a *deep embedding* [49], meaning it provides a core API that internally constructs syntax trees, using host-level evaluation in Haskell as a kind of macro system. This approach allowed us develop Scoria much more rapidly than SSLang, and provided a platform for experimenting with language features and testing our language runtime.

3.2.1 SSM Primitives

Scoria programs are Haskell expressions that, when evaluated, produce C code that can be linked against the same SSM runtime library SSLang uses. In other words, Scoria uses Haskell as a metaprogramming environment to build SSM programs. Figure 3.11 lists the names and

```
ref     :: Exp a -> SSM (Ref a)                  -- New scheduled variable
deref   :: SSMType a => Ref a -> Exp a           -- Read value
changed :: Ref a -> Exp Bool                     -- Was written?
assign  :: Ref a -> Exp a -> SSM ()              -- Assignment
after   :: Exp Time -> Ref a -> Exp a -> SSM ()  -- Delayed assignment
wait    :: Waitable a => a -> SSM ()             -- Block until assignment
par     :: [SSM ()] -> SSM ()                    -- Concurrency
```

Figure 3.11: SSM primitives in Scoria.

types of Scoria's core API functions.[8] Each is a Haskell function that manipulates and constructs some kind of Scoria program fragment, stored internally as an abstract syntax tree—this reified data structure is what makes Scoria a *deep* embedding rather than a shallow one. The types of Scoria's API functions ensure users can only construct structurally and semantically (type-safe) valid syntax trees.[9]

Scoria uses several type constructors to categorize the various kinds of program fragments. For instance, an `Exp a` is a Scoria value of type `a`; scheduled variables storing some type `a` appear as `Ref a`. The most important of Scoria's type constructors is `SSM a`, which represents Scoria *computations* that produce a result of type `a`.[10] Scoria declares `SSM` to be a Haskell monad instance, so blocks of `SSM` terms can be composed using Haskell's `do`-notation, as seen in Figure 3.10. For instance, scheduled variables of type `a` are created using the `ref` function, which returns an `SSM (Ref a)`. The reference to the scheduled variable is "wrapped inside" an `SSM` computation, but can be extracted and bound to a variable inside a `do`-block. For example:

```
do                  -- The do keyword begins a monadic block
    x <- ref 10     -- Bind result (a scheduled variable) to x
    assign x 20     -- We can now use x in other operations
```

In Scoria, the meaning and interface of `assign` and `after` closely follow the SSM primitives they are based on; both are presented as side-effectful statements that "return" the unit value

[8] I have made some cosmetic changes to Scoria here for clarity and consistency with the rest of this dissertation. In the original presentation of Scoria [63], `ref` was `new`, `assign` was `(<~)`, and `par` was `fork`.

[9] In Haskell's type syntax, identifiers beginning with a lowercase letter designate type variables, while those beginning with an uppercase letter refer to types (or type classes): the type "`Exp a`" means "for any type `a`, an expression (`Exp`) of type `a`." Right arrows denote (curried) functions: "`A -> B -> C`" means "a function that takes an `A` and a `B` and returns a `C`."

[10] Here, "computation" means roughly the same thing as "statement" except statements do not usually return a result. To be more precise, a statement is a computation that returns a unit value, i.e., `SSM ()`.

```
class Waitable sv where
    wait :: sv -> SSM ()
instance Waitable (Ref a) where
    wait = ···
instance Waitable (Ref a, Ref b) where
    wait = ···
```

Figure 3.12: Scoria's `Waitable` type class.

`()` as their result. However, `deref` and `changed` (Scoria's version of SSM's `since` primitive; see Section 2.3.1) are *expressions* that can be used in any context expecting an `Exp`. This is what allows us to write the `sum` function's assignment statement (line 4 of Figure 3.10) in a single line, without needing to bind the results of `deref x` and `deref y` in separate statements.

Scoria's `wait` function supports blocking on one or more scheduled variables. Haskell does not support *variadic* functions—functions that support a variable number of arguments—but Scoria approximates these using the `Waitable` type class, shown in Figure 3.12. Haskell's *type classes* [43] allow us to overload a function with *instances* of that type class. We exploit this feature and declare a separate `Waitable` instance for each tuple size (up to a reasonable bound, i.e., 64), so that we can pass a tuple of any size (or a single scheduled variable) to `wait`.[11] A tuple literal still counts as a single argument, but *looks* like multiple arguments, and allows us to write valid Scoria statements like this:

```
wait x              -- Wait for a single scheduled variable
wait (x, y, z)      -- Wait for three scheduled variables at the same time
```

Scoria's `par` function also supports executing a variable number of `SSM ()` statements in parallel. `par`'s interface is simpler than `wait`'s interface because it requires that all parallel statements to have the same type. Thus, `par` statements can use Haskell's list syntax to receive a (homogeneously typed) list of computations:

```
par [x]             -- Evaluate a single expression (same as a function call)
par [x, y, z]       -- Evaluate three expressions in parallel
```

Using lists like this does not work for `wait` statements because Scoria must support waiting on scheduled variables of different types.[12]

[11] The prefix "`Waitable a =>`" is a constraint on the type variable `a`; it means "`a` must be an instance of `Waitable`."
[12] Haskell's `ExistentialQuantification` extension lets us to overcome this limitation, but adds complexity.

3.2.2 Being a Good Host (Language)

Although Scoria was primarily designed to rapidly prototype our programming model and runtime, our secondary goal was to explore the ergonomics of sparse synchronous programming. We wanted to go beyond a library for "bare" syntax trees and see whether it is possible to build SSM systems with the "look and feel" of its host language, Haskell. We have already discussed Scoria's monadic `SSM` API in the previous section; this section will cover a few more "Haskell embedding tricks" that help Scoria look more like a language than library or framework.

Scoria supports integers of varying size and signedness, and can perform arithmetic on them. Scoria declares them to be instances of Haskell's `Num` type class, so they can take advantage of Haskell's already-overloaded numeric literal and arithmetic syntax. These instances allow us to write expressions such as `n-1` in Figure 3.10, rather than invent our own syntax for integers and subtraction. Under the hood, `n-1` elaborates to the following program fragment:

```
Exp (BinOp (typeOf n)                -- Binary operator, returns same type as n
           OpMinus                   -- Operator: minus
           n                         -- Left operand: n (already an Exp)
           (Exp (Lit TInt32          -- Right operand: literal of type TInt32
                     (LInt32 1))))   -- ... and value 1
```

The difference between this Scoria data structure and a "native" Haskell term is that the former can be translated to C, whereas the latter can only be evaluated in Haskell itself.

However, this approach does not work for all numeric operators because some standard type classes are not as flexible as `Num`. For instance, Haskell's comparison operator < has the type:

```
(<) :: Ord a => a -> a -> Bool
```

Defining `Ord` instances does not work here because < must return a `Bool`, and not an `Exp Bool`. Though it is possible to shadow Haskell's definition of < with our own, we felt it was more expedient and almost as self-explanatory to define our own operator, `<.` (used in Figure 3.10):

```
(<.) :: (Num a, SSMType a) => Exp a -> Exp a -> Exp Bool
```

The `SSMType` constraint, which also appears in `deref`'s type signature, ensures that Scoria can extract a type annotation (e.g., `TInt32`) that it can embed in the constructed program fragment.

Scoria also uses other mechanisms to overload Haskell syntax; most of these mechanisms come from GHC, the Haskell compiler we use to evaluate Scoria's host language. Haskell's conditional expressions (`if-then-else`) are usually hardwired to test `Bool`-typed conditions, and cannot be reconfigured to generate user-defined terms. However, Scoria removes this limitation using a GHC extension that is enabled by the following incantation:

```
{-# LANGUAGE RebindableSyntax #-}
```

With the `RebindableSyntax` extension, `if-then-else` expressions resolve to Scoria's own definition, which has the following type signature:

```
ifThenElse :: Exp Bool -> SSM () -> SSM () -> SSM ()
```

and internally constructs the appropriate Scoria program fragment. For instance, the conditional statement in Figure 3.10 is equivalent to:

```
ifThenElse (n <. 2)
           (after (secs 3) r 1)
           (do f1 <- ref 0
               f2 <- ref 0
               par [fib (n-1) f1, fib (n-2) f2, sum f1 f2 r])
```

Scoria defines a `routine` "keyword" to help it distinguish between regular Haskell functions and Scoria routines. Both kinds of definitions share the same syntax, but in the latter case, Haskell must construct a reified syntax tree rather directly evaluating the function body. This distinction is also necessary because it creates a layer of indirection that allows Scoria programs to perform recursion without endlessly inlining the routine body.

Routine definitions cannot be trivially embedded in Haskell because it requires each routine definition to reflect on its own name and the names of its arguments. An earlier iteration of Scoria expected users to explicitly convey these names as Haskell strings:

```
f = make_routine "f" ["x", "y"] (\x y -> body)
```

This `make_routine` function had an awkward interface with plenty of unnecessary repetition. To overcome the syntactic redundancy, Scoria leverages GHC's plugin support to define `routine`

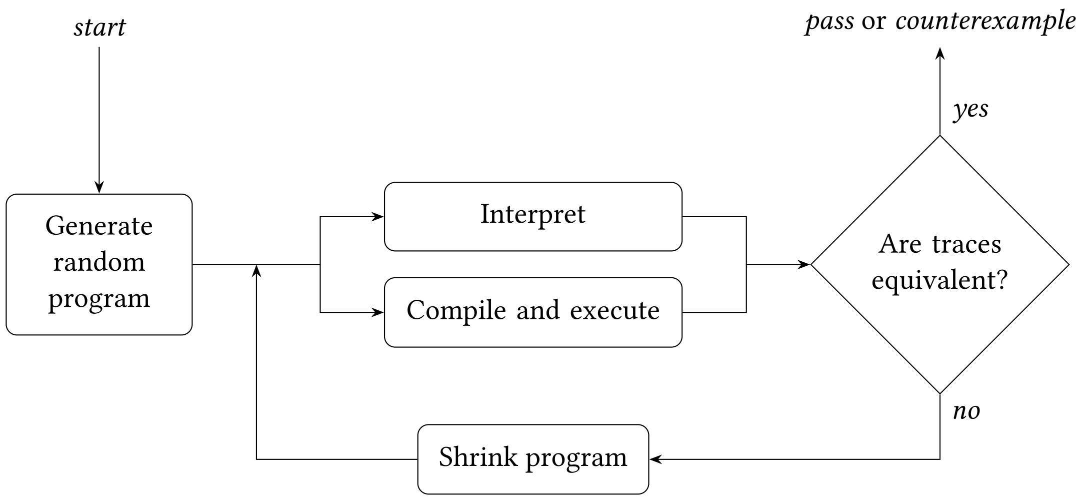


Figure 3.13: How we tested the semantic correctness of generated Scoria programs.

as a synthetic "keyword" that annotates function definitions. Our plugin looks for invocations of **`routine`** and replaces it with an invocation of `make_routine`. For instance, this source-to-source transformation produces the above boilerplate from the following:

```
f x y = routine body
```

This **`routine`** operator also appears in our Figure 3.10 example and ensures the recursive `fib` definition does not construct an infinite syntax tree.

Scoria has several limitations we did not address because we prioritized primary goal of rapid prototyping over providing a comprehensive and robust set of language features. Many of these limitations come from its implementation as a deep embedding: for instance, Scoria is limited to manipulating integer and Boolean values, and does not support user-defined algebraic data types or closures. These features may be achieved via more "heavyweight" embedding techniques [34, 74, 92] which we leave to be explored in future work.

3.2.3 Testing

The functions and language extensions discussed in the previous sections allow users to build Scoria programs that visually resemble Haskell programs, but construct reified syntax trees beneath the surface. Scoria provides a compiler that translates this syntax tree into C code which we can compile, link, and run with our SSM runtime (see Chapter 4). Scoria also includes an SSM implementation that interprets this syntax tree in Haskell itself, which we use as a reference to validate our compiler's output.

To test our system, we used QuickCheck [21], a property-based testing framework for Haskell. QuickCheck generates random values according a user-defined generator, which it tests against user-specified properties; in our case, the random values we generated were Scoria programs. If a property is falsified, QuickChecks shrinks the test case until it discovers an approximately "minimal" failing test case—a *counterexample* to the property—according to a shrinker that is also user-defined. Shrinking is important for diagnosing the property failure, because randomly generated programs are typically too large to comprehend.

We tested two properties: the first is that generated C code compiles and runs without memory errors or leaks (according to Valgrind [90]). This property helped us find a bug related to our initial event queue implementation: when a scheduled variable was freed, pending scheduled events could retain stale pointers to those freed variables. Eventually, the runtime system would attempt to perform updates using these stale pointers, corrupting the program state. We fixed this bug by having scheduled variables remove themselves from the event queue when freed.

We also tested semantic correctness of the generated program, according to the flow chart shown in Figure 3.13. For test builds, our compiler adds logging statements to the generated code to produce an event trace, which we compare to the trace from our reference interpreter. Although this property mostly helped us catch interpreter bugs, it also helped identify a critical issue with our code generation strategy. We initially implemented directly translated Scoria signed integers to C signed integers. However, when QuickCheck generated test cases that branched on overflowing arithmetic, the generated code would behave unexpectedly due to C's undefined overflow behavior. This led Scoria—and later SSLang—to adopt a compilation scheme based on C's better-defined unsigned integers.

```lua
function sum(x, y, z)
    ssm.wait {x, y}
    z.val = x.val + y.val
end

function fib(n, r)
    if n < 2 then
        r:after(ssm.secs(3), { val = 1 })
    else
        local f1, f2 = ssm.Channel {}, ssm.Channel {}
        ssm.wait {
            ssm.spawn(fib, n - 1, f1),
            ssm.spawn(fib, n - 2, f2),
            ssm.spawn(sum, f1, f2, r),
        }
    end
end
```

Figure 3.14: A contrived Fibonacci example adapted from Figure 2.3, implemented using ssm-lua. `par` is supplanted by `spawn`; scheduled variables are replaced by "channel tables," allocated using the `Channel` constructor function.

3.3 ssm-lua: a Shallow Embedding of SSM in Lua

From an implementor's perspective, standalone and deeply embedded languages like SSLang and Scoria give us a lot of control over how programs behave, because we can explicitly design, manipulate, and analyze the data structures that represent a program at compile time. Emitting C code and linking against a purpose-built runtime library also lets us fine-tune a program's memory layout and footprint at runtime. However, this low-level approach is laborious and error-prone, leaving us with the responsibility of managing each stack frame and heap object. New languages also suffer from a lack of libraries and tooling that established languages enjoy.

To experiment with an alternative approach, we implemented SSM as a library for Lua, a lightweight scripting language [57]. Figure 3.14 shows the Fibonacci example from Chapter 2 written with our library, ssm-lua. In this work, we made a few changes to our original programming model to better suit Lua's programming idioms. Our library builds on existing Lua features like tables and coroutines to implement SSM concepts like processes and scheduled variables.

ssm-lua is independent of the SSM runtime that SSLang and Scoria use. Instead, it is written in just under 1000 lines of pure Lua, and is compatible with Lua versions 5.1 to 5.4. In contrast to

Scoria, ssm-lua is a *shallow* embedding [49] and does not require a separate runtime.

As a Lua library, ssm-lua takes advantage of its host language's existing features, libraries, and ecosystem, and does not require modifications to the Lua runtime. For instance, Lua's incremental garbage collector relieves a major implementation burden, while its standard library, module system, and robust C FFI capabilities allow ssm-lua to integrate with existing code.

3.3.1 Channel Tables

Tables are Lua's implementation of associative arrays [7], and they appear everywhere in the language as a data structure for mapping keys to values. For instance, they are used as (1-indexed) arrays when the keys are integers, and as packages when the keys are the names of library functions (for instance, ssm-lua is accessed via the table `ssm` in Figure 3.14). Their behavior can also be extended to support operating overloading and class inheritance using metatables.

ssm-lua uses this mechanism to implement *channel tables*, which play the role of SSM's scheduled variables. They behave like regular tables and associate non-`nil` keys with non-`nil` values, but also support SSM's delayed assignments (**after**) and blocking (**wait**). Figure 3.14 uses channel tables in place of the scheduled variables in Figure 2.3; Figure 3.15 shows a simpler use of channel tables, to implement the `sleep` and `timeout` routines from Figure 2.2.

Channel tables are created by the `Channel` constructor, with their fields initialized according to the table given to this constructor; those fields are accessed using Lua's dot or index notation:

```
local tbl = ssm.Channel { foo = 10 }
assert(tbl.foo == 10)         -- Access foo using dot notation
assert(tbl["foo"] == 10)      -- Equivalent index notation
```

Channel tables can also be updated like regular Lua tables by assigning to some non-`nil` key:

```
tbl.foo = 20                  -- Overwrite existing entry
tbl.bar = 30                  -- Define new entry
assert(tbl.foo == 20 and tbl.bar == 30)
```

Updating a channel table like this is an instantaneous assignment á la SSM, and unblock all lower processes waiting on that channel table.

```
function ssm.sleep(d)
    local x = ssm.Channel {}
    x:after(d, { go = true })
    ssm.wait(x)
end
```

```
function ssm.timeout(d, c)
    local x = ssm.Channel {}
    x:after(d, { go = true })
    ssm.wait(x, c)
end
```

```
function ssm.measureDuration(e)
    local timer = ssm.Channel { begin = true }
    e()
    return timer:since("begin")
end
```

Figure 3.15: Implementation of `sleep`, `timeout`, and `measureDuration` in ssm-lua, using channel tables. The logic is the same as in Figure 2.2, but ssm-lua requires us to choose arbitrary keys like `go` and `begin` when invoking channel table methods, to maintain consistency with how regular Lua tables behave.

For delayed assignments, the `after` method schedules updates to the specified keys of a channel table:

```
tbl:after(ssm.msec(10), { foo = 40 })
assert(tbl.foo == 20)       -- Delayed assignment does not happen until later
ssm.wait(tbl)               -- Wait for the update; only foo is updated
assert(tbl.foo == 40 and tbl.bar == 30)
```

The delayed assignment `{ foo = 24 }` is specified as a table literal in the last argument to `after`, but it can also be constructed dynamically, e.g.:

```
local update = {}
update.foo = 40
tbl:after(ssm.msec(10), update)
```

Unlike SSM's scheduled variables, delayed (and instantaneous) assignments are made to individual keys, rather than to the table as a whole.[13] This is why the `sleep` and `timeout` functions in Figure 3.15 need to assign to an arbitrary key, `go`.

We can schedule multiple assignments for the same time by adding more entries to the table passed to `after`:

```
tbl:after(ssm.msec(10), { foo = 50, baz = 60 })
assert(tbl.baz == nil)      -- baz is still not defined yet
ssm.wait(tbl)               -- Block until foo and baz are both updated
assert(tbl.foo == 50 and tbl.bar == 30 and tbl.baz == 60)
```

[13] In Lua, writing `tbl = val` means something very different than `tbl.key = val`: it binds the variable `tbl` to refer to `val`, rather than updating the value that `tbl` refers to.

Channel tables also allow us to schedule multiple delayed assignments, provided those keys do not overlap:

```
tbl:after(ssm.msec(10), { foo = 70 })
tbl:after(ssm.msec(20), { bar = 80 })
tbl:after(ssm.msec(20), { baz = 90 })

ssm.wait(tbl)          -- Block until write to foo
assert(tbl.foo == 70 and tbl.bar == 30 and tbl.baz == 60)

ssm.wait(tbl)          -- Block until writes to bar and baz
assert(tbl.foo == 70 and tbl.bar == 80 and tbl.baz == 90)
```

Each key may only have one outstanding assignment; scheduling another assignment on the same key overwrites any existing one. However, programs can overcome this limitation and schedule an unbounded number of events by assigning to a unique key each time.

Like scheduled variables, channel tables allow us to query how long it has been "`since`" an update. Figure 3.15 shows an application of this method to implement the measureDuration function from Figure 2.2. In ssm-lua, the last-written time is tracked separately for each key (to be consistent with the assignment interface), so the `since` method takes a parameter indicating which key should be queried. We use a string literal to name the key since dot-notation (`tbl.key`) is just syntactic sugar for index-notation (`tbl["key"]`).

Implementation

ssm-lua `Channel` constructor, shown in Figure 3.16, uses Lua's *metatable* mechanism to implement the behavior of SSM's scheduled variables. Metatables are Lua tables that can be attached to other tables[14] to hold metadata. Metatables can also hold *metamethods* which overload certain table operations. The metatable, `chan`, is constructed in line 2 and initialized throughout the rest of the `Channel` constructor; it is attached to an empty "proxy" table in in line 5, which is later

[14] Metatables can also be attached to Lua *userdata* objects, which are typically allocated via Lua's C API to store arbitrary data. Though implementing channel tables with userdata would be more efficient, they can only be allocated in Lua via its undocumented `newproxy` function. To keep this discussion simple, we will use empty tables in lieu of userdata.

```
 1 function ssm.Channel(init)
 2     local chan = {}                       -- Used as a metatable
 3
 4     local proxy = {}                      -- Only used to "carry" a metatable
 5     setmetatable(proxy, chan)             -- Attach metatable to proxy
 6
 7     chan.value = {}                       -- "Shadow table" where entries are stored
 8     chan.last_written = {}                -- When each entry was last assigned
 9     chan.later_value = {}                 -- Where delayed assignments are stored
10     chan.later_time = {}                  -- When delayed assignments should happen
11     chan.triggers = {}                    -- What to run when assigned
12     chan.earliest = math.huge             -- Earliest scheduled update
13
14     chan.value.after = channel_after      -- Attach after method
15     chan.last_written.after = get_current_time()
16     chan.value.since = channel_since      -- Attach since method
17     chan.last_written.since = get_current_time()
18
19     for key, val in pairs(init) do        -- Populate shadow table with initial values
20         chan.value[key] = val
21         chan.last_written[key] = get_current_time()
22     end
23
24     chan.__index = chan.value             -- Read entries from shadow table
25     chan.__newindex = channel_assign      -- Overload (instantaneous) assignment
26
27     return proxy                          -- Users only interact with the proxy
28 end
```

Figure 3.16: Constructor for ssm-lua's channel tables.

returned in line 27. This proxy table is what programs actually interact with, though all of its channel table behavior is implemented via the metatable `chan` attached to it.

Most of the fields of a channel table's metatable carry metadata. They roughly mirror SSLang's `sv_t` (Figure 3.7) and are initialized in lines 7–12. Channel tables do not have any scheduled updates when they are just created, so the `later_value`, `later_time`, and `triggers` fields (lines 7–11) are initialized empty. The `earliest` field (line 12) is unique to channel tables, and is used to maintain the position of the channel table in the event queue in the presence of multiple delayed assignments. It is initialized to `math.huge` (positive infinity) to represent that nothing has been scheduled yet.

Channel tables maintain their entries in a separate "shadow" table, `value`. The `Channel` constructor populates this shadow table's initial values based on the entries in `init`, which it iterates over in lines 19–22. It also attaches the `after` and `since` methods to the shadow table as regular entries (lines 14–17). Lua methods are table entries whose value is a function which receives `self` as its first parameter, and `tbl:after(d, v)` is just shorthand for writing `tbl.after(tbl, d, v)`. All entries of the shadow table have a corresponding entry in `last_written`, which maintains when each entry was last assigned. In the `Channel` constructor, these are all initialized to the current logical time.

Recall that the channel table given to a user is actually proxy whose metatable implements all of its behavior. When a program indexes into a channel table (e.g., `proxy[key]`) the entry is actually read from or written to the underlying shadow table maintained in its metatable (e.g., `getmetatable(proxy).value[key]`). ssm-lua implements this behavior by overloading the `__index` and `__newindex` metamethods. In line 24, setting `__index` to the shadow table `value` causes all reads to be looked up from `value`. In line 25, `__newindex` is set to the `channel_assign` function, which implements instantaneous assignment à la SSM: when a channel table is assigned, it writes to the corresponding entry in the shadow table, and schedules sensitive lower-priority processes (in `triggers`) for execution.

3.3.2 Multi-way Waiting

As in SSM, ssm-lua'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 ssm-lua's only directly blocking primitive; processes may also block indirectly by calling a function that calls `wait`.

As with scheduled variables, any assignment to a channel table (instantaneously or delayed) will unblock a `wait`, including those that do not change the existing value:

```
local tbl = ssm.Channel { go = true }
tbl:after(ssm.msec(10), { go = true })
ssm.wait(tbl)            -- Unblocks after 10ms, even though go has the same value
```

Unlike the `wait` primitive from SSMΛ (and SSLang), ssm-lua's `wait` also supports blocking until *all* channel tables have been assigned. The `wait` function accepts a variable number of arguments, and its general form is `wait(`w_1`, ..., `w_n`)` where each w denotes a *wait spec*; `wait` blocks until any of its n wait specs are satisfied. A wait spec is either a single channel table c or a (non-empty) Lua array of k channel tables `{ `c_1`, ..., `c_k` }`, and is satisfied once all specified channel tables have been updated. (Specifying a single channel table c is equivalent to giving a single-element array `{ `c` }`.) For example, to block until both `a` and `b` are updated, or until `c` is updated, we write:

```
ssm.wait({a, b}, c)
```

Stated more succinctly, ssm-lua supports both disjunctive and conjunctive unblocking conditions, as well as a combination of the two in disjunctive normal form; see Section 2.3.3 for more discussion about this design choice.

Upon unblocking, the variadic `wait` function returns as many Booleans as the number of arguments it was called with. Each returned Boolean indicates which wait spec was met, so at least one of the returned Booleans is true:

```
local ab_written, c_written = ssm.wait({a, b}, c)
assert(ab_written or c_written)
```

Lua syntax allows us to omit parentheses for function calls when the only argument is a table literal,[15] so we can write a purely conjunctive `wait` with or without parentheses:

```
ssm.wait({a, b, c})              ssm.wait {a, b, c}
```

The two statements are equivalent and both mean "block until `a`, `b`, and `c` have all been updated."

Meanwhile, a purely disjunctive `wait` can be written just as concisely, because ssm-lua treats individual channel tables as single-element wait specs:

```
ssm.wait({a}, {b}, {c})          ssm.wait(a, b, c)
```

Both statements mean "block until either `a`, `b`, or `c` have been updated." Figure 3.14 features purely conjunctive `wait` statements, on lines 2 and 11–15, while the `timeout` function from Figure 2.2 uses a purely disjunctive `wait` in line 9.

```
 1 function ssm.wait(...)
 2     local this = get_running_process()     -- Retrieve handle to current process
 3     local specs = gather_specs(...)        -- Pack variadic arguments
 4     sensitize_all(specs, this)             -- Add this process to triggers
 5     while not any_met(specs, this) do      -- Until at least one wait spec is met
 6         coroutine.yield()                  -- ... yield execution to scheduler
 7     end
 8     desensitize_all(specs, this)           -- Remove this process from triggers
 9     return specs_written(specs)            -- Unpack variadic return values
10 end
11
12 local function process_resume(this)
13     local prev = get_running_process()     -- Save current process as previous
14     set_running_process(this)              -- Set next process as current
15     coroutine.resume(this.co)              -- Resume execution of coroutine
16     for i=#this.deferred, 1, -1 do         -- In last-in, first-out order
17         process_resume(this.deferred[i])   -- ... run each deferred process
18         this.deferred[i] = nil             -- ... and remove it from the list
19     end
20     set_running_process(prev)              -- Restore previous process
21 end
```

Figure 3.17: Implementation of `wait` in ssm-lua, using Lua's coroutines. When `wait` yields the running coroutine (line 6), execution continues in the `process_resume` helper function (line 15) where that coroutine was run from.

Implementation

ssm-lua uses Lua coroutines [27] to implement the suspension and resumption of SSM processes. Processes suspend when they call the `wait` function, whose implementation is shown in Figure 3.17 alongside the `process_resume` helper function, which ssm-lua uses to run and resume processes. The `process_resume` function calls Lua's built-in `coroutine.resume` to resume a suspended process (line 15); when `wait` calls `coroutine.yield` to suspend a process (line 6), execution returns the `coroutine.resume` call site in `process_resume`.[16] Lua coroutines are *stackful*, so `coroutine.yield` (hence ssm-lua's `wait`) works at arbitrarily deep levels of the Lua call stack.

ssm-lua's `wait` implementation calls `coroutine.yield` in a loop because it is possible for a

[15] This is also why we can write `Channel { ... }` instead of `Channel({ ... })`; `Channel` is just a regular Lua function that constructs channel tables.

[16] Lua's coroutine API is *asymmetric*: `coroutine.resume` expects a coroutine object as its argument because it needs to know what to resume, but `coroutine.yield` does not and returns execution to wherever the yielding coroutine was resumed from.

coroutine to be resumed when its wait specs are only partially met. For example, a process that calls `wait {a, b}` will be resumed when only a is assigned, even though it should block until both a and b have been updated. If that happens, the `any_met` condition in line 5 will return `false` and cause `wait` to suspend again, until b has also been assigned.

3.3.3 Concurrency

ssm-lua provides two primitives for concurrent function calls, `spawn` and `defer`. They both create a *child* process to execute the specified function call. The priority of the child process 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. Thus, calling `spawn` multiple times will create processes with successively lower priorities:

```
ssm.spawn(foo)          -- Highest priority
ssm.spawn(foo)          -- Next highest priority
...                     -- Lowest priority (parent)
```

`defer` is the dual of `spawn`: the created child is given the next lowest priority. It does not run its first instant until the parent suspends or terminates. Calling `defer` multiple times will produce the opposite pattern of `spawn`, with processes of successively higher priorities:

```
ssm.defer(foo)          -- Lowest priority
ssm.defer(foo)          -- Next lowest priority
...                     -- Highest priority (parent)
```

The general form of a `spawn` (or `defer`) call is `spawn(`f`, `a_1`, ..., `a_n`)`. The first argument, f, is a *synchronous function* that the newly created process will run. A synchronous function is a Lua function that may invoke ssm-lua primitives, and may only run in coroutines managed by ssm-lua (using `process_resume`; see Figure 3.17). f is followed by zero or more arguments a, passed to f in the newly created process. For example, line 12 of Figure 3.14 spawns a new process running `fib(n - 1, f1)`.

Since anonymous closures are first-class values in Lua, we can pass them directly to `spawn` and `defer`. These closures are created using Lua's `function` keyword, can captures variables

```
 1  function ssm.clock(period)
 2      local clk = ssm.Channel {}
 3      ssm.spawn(function()
 4          while true do
 5              clk:after(period, { go = true })
 6              ssm.wait(clk)
 7          end
 8      end)
 9      return clk
10  end
```

Figure 3.18: An ssm-lua function that constructs a `clock` object. The clock is implemented as a channel table that is updated at the specified period by a spawned process.

from their enclosing scope. For instance, the closure in lines 3–8 of Figure 3.18 captures the `clock` function's argument `period` and local variable `clk`.

`spawn` and **`defer`** create each child process with a *promise channel*, which is returned to the parent. This channel table behave like regular channels, and allows the parent to wait for its children and receive their return values. When a child process terminates, its return values are (instantaneously) assigned to its promise channel.

Promise channels allow ssm-lua to concisely recover the behavior of SSMΛ and SSLang's **par**, which blocks until all child processes have terminated. Any instance of **par** (p_1, ..., p_n) can be rewritten as:

ssm.wait { **ssm.spawn**(p_1), ..., **ssm.spawn**(p_n) }

The `fib` function from Figure 3.14 uses this pattern in lines 11 to 15.

Lua functions may return zero or more values. Those are assigned to promise channels positionally, so each return value is written to a successive index, starting from 1:

```
local rc = ssm.spawn(function()
    ssm.sleep(ssm.msecs(3))
    return "abc", 123
end)
ssm.wait(rc)
assert(rc[0] == true and rc[1] == "abc" and rc[2] == 123)
```

ssm-lua always assigns `true` to index 0 to ensure an assignment takes place, even if the child process does not return anything. The `fib` function from Figure 3.14 does not use this feature, to be consistent with the example it is based on from Figure 2.3.

```
function ssm.spawn(func, ...)
    local this = get_running_process()
    local promise = ssm.Channel {}          -- This will be returned to caller
    local prio = this.prio                  -- New priority comes before current
    this.prio = prio:insert_after()
    local proc = process_new(func, { ... }, promise, prio)
    process_resume(proc)                    -- Run first instant of new process
    return promise
end

function ssm.defer(func, ...)
    local this = get_running_process()
    local promise = ssm.Channel {}          -- This will be returned to caller
    local prio = this.prio:insert_after()   -- New priority goes after current
    local proc = process_new(func, { ... }, promise, prio)
    process_enqueue(proc)                   -- Schedule process for later
    return promise
end
```

Figure 3.19: Implementation of `spawn` and `defer`.

Implementation

ssm-lua's implementation of `spawn` and `defer` are shown in Figure 3.19. Where possible, ssm-lua uses Lua's own call stack to instead of touching the scheduler's run queue unnecessarily; doing so avoids the overhead of maintaining that priority queue. Rather than yielding to the scheduler, `spawn` directly runs the first instant of its newly created process by calling the `process_resume` helper from Figure 3.17. Similarly, `defer` adds its newly created process to the running process's `deferred` list; after it eventually yields to `process_resume`, those deferred processes are run from that list in last-in, first-out order (lines 16–19).

Unlike in SSLang, ssm-lua's synchronous function calls (e.g., just calling `sleep(d)` or `fib(n, r)`, without `spawn` or `defer`) require no special treatment since they are just regular Lua function calls; its `wait` function works from any point in the Lua call stack. By contrast, SSLang treats synchronous function calls as unary `par` statements to ensure processes can yield at any point of the call stack, at the expense of allocating more processes.

Each SSM process must be assigned a unique priority to ensure that they are totally ordered.[17] SSLang assigns priorities to processes according to their position in the process tree, but this

[17] This is known as the "order maintenance problem" in the algorithms literature.

scheme limits the depth of the process tree to the number of bits given to priorities (i.e., 32). It also relies on processes being organized as a tree, where child processes always terminate before their parent. This structure follows naturally from SSLang's **`par`** primitive, but ssm-lua is more flexible and allow children to live longer than their parents.

To overcome these limitations, ssm-lua implements priorities using Dietz & Sleator's *tag-range relabeling* algorithm [6, 29]. Their algorithm 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. Tag-range relabeling guarantees up to $2^{N/2} - 1$ distinct priorities, where N is the number of bits used to represent integers. Lua typically uses IEEE 754 double-precision floating-point numbers [54], which theoretically allows ssm-lua to support up to $2^{52/2} - 1$ processes—Lua will likely run out of memory before then.

In ssm-lua's implementation of priorities, the `insert_after` method constructs a new priority that is inserted immediately after the priority it is called on. The **`spawn`** and **`defer`** functions in Figure 3.19 use this method to assign a new priority relative to that of the current running process. To make up for the lack of an `insert_before` method, the **`spawn`** function gives the current process's priority to the new process, and adopts the next highest priority created by `insert_after`.

Chapter 4: Implementing a Sparse Synchronous Language Runtime

To support SSLang and Scoria, we developed a language runtime library that ensures processes are run at the right time and in the right order. It consists of a platform-agnostic cooperative *scheduler*, which manages processes and assignment events, and a platform-specific *driver*, which tries to reconcile the SSM system with its environment.

The runtime scheduler implements an SSM system's logical behavior, which must be consistent from platform to platform. The scheduler exposes a "tick" function that advances logical time between instants, and coordinates the computation taking place within each instant. It performs outstanding delayed assignments that were scheduled from earlier instants, and determines which processes to run and what order to run them in.

The driver uses with its platform's timers to manage the system's physical timing, and is responsible for calling the scheduler's tick function at an appropriate (physical) time. Furthermore, it allows the SSM program to perform timed I/O. As depicted in Figure 4.1, the driver conveys external events to the program via scheduled variables that represent the environment's inputs and outputs. Though these input and output events carry logical timestamps, the driver times them in accordance with the platform's physical timer so that they accurate reflect the environment.[1]

[1] Kyle J. Edwards helped us develop timestamp peripherals to implement timed I/O on the RP2040 platform.

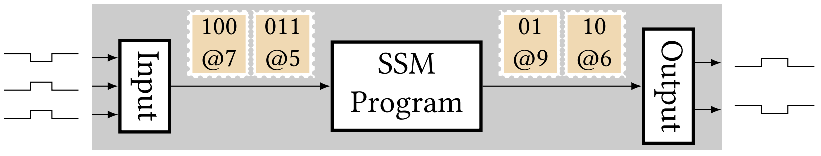


Figure 4.1: How SSM programs perform I/O using input and output variables. A peripheral interprets changes on inputs as timestamped events, which passed to our SSM runtime and reflected in input variables. Assignments by the program to output variables are also conveyed as timestamped events to a peripheral that manipulates outputs.

```
globals
  ▹ $CurrentTime$   // Current logical timestamp
  ▹ $EventQueue$    // Priority queue of scheduled variables, ordered by later_time
  ▹ $RunQueue$      // Priority queue of processes, ordered by priority

procedure UPDATE($sv$)                          // Perform delayed assignment at current instant
    requires $sv$.last_updated < $sv$.later_time  // Time should only advance forward
    requires $sv$.later_time = $CurrentTime$      // Delayed assignment should be scheduled for now
    set $sv$.value        ⟵ $sv$.later_value      // Overwrite current value with scheduled value
    set $sv$.last_updated ⟵ $CurrentTime$         // Overwrite last updated timestamp with now
    set $sv$.later_time   ⟵ ULONG_MAX             // Mark scheduled asssignment as done
    for each $p$ in $sv$.triggers                 // Schedule each process waiting on sv
        if $p$ is not in $RunQueue$               // …that is not already scheduled
            enqueue $p \rightarrow RunQueue$      // …for execution this instant

procedure TICK                                  // Run a process for the next instant
    requires $EventQueue$ is not empty            // There should be events to process
    requires $CurrentTime$ < earliest in $EventQueue$  // Time should only advance forward
    set $CurrentTime$ ⟵ earliest in $EventQueue$  // Phase 1: jump to next earliest event
    while $EventQueue$ is not empty and           // Phase 2: for each scheduled variable with
            $CurrentTime$ = earliest in $EventQueue$  // …a delayed assignment at the current instant
        let $sv$ ⟵ dequeue $EventQueue$           // Remove it from the event queue
        call UPDATE($sv$)                         // Execute the delayed assignment
    while $RunQueue$ is not empty                 // Phase 3: for each scheduled process
        let $p$ ⟵ dequeue $RunQueue$              // Remove it from the run queue
        run step function of $p$                  // …and run it for an instant
```

Figure 4.2: Pseudocode for the SSM runtime's tick function, and for performing delayed assignments.

4.1 Scheduling Events and Processes

The SSM runtime's platform-agnostic scheduler maintains two priority queues, the event queue and the run queue. The *event queue* holds scheduled to be updated, ordered according to their `later_time` fields, which stores the delayed assignment is scheduled for. Meanwhile, the *run queue* holds the activation records of functions (processes) scheduled to run in the current instant, ordered by increasing `priority` fields. We implement both as binary heaps whose maximum size can be determined if the program's dynamic call graph can be analyzed statically.

Our runtime's event queue corresponds to the event queue ϕ from our formal semantics (Section 2.2). The run queue avoids unnecessary work for processes that do not need to resume: while the S-TICK rule prescribes reducing every redex of the running program, including **check** expressions that will immediately block again, the run queue avoids "busy waiting" by maintaining only the set of *active* processes that will unblock and run in the current instant.

The SSM runtime scheduler exposes a `tick` function that runs the system for an instant; its pseudocode shown in Figure 4.2.[2] This function works in three phases: advancing logical time, performing all the assignment events queued for the current instant, then running every process in the run queue in priority order.

In the first phase, the `tick` function advances logical time to the earliest timestamp in the event queue. Doing so skips over *inactive* instants where no computation needs to take place, which is why we designed our model of time to be "sparse."

In the second phase, performing an event consists of removing the scheduled variable at the front of the event queue provided it is scheduled for the current instant, updating its `value` and `last_written` fields, and then adding each process waiting on the variable (held in its list of `triggers`) to the run queue if it is not already there. This phase ends when there are no pending events on the queue for the current time instant. Note that each scheduled variable's list of triggered processes is not modified during this phase: the processes themselves are exclusively responsible for managing their triggers.

In the third phase, the process with the lowest priority number[3] is removed from the run queue and its step function invoked. The step function, in turn, may cause processes with equal or higher priority numbers to be added to the ready queue, either by performing (parallel) function calls, or by assigning to a scheduled variable that triggers other waiting processes to run.

The scheduler will terminate unless some process refuses to suspend. Processes may perform multiple recursive calls or loop iterations in a single instant; SSM requires that they terminate in order for logical time to advance, but SSLang does not enforce this (nor do Scoria or ssm-lua).

[2] ssm-lua is implemented independent of the SSM runtime, but its tick function follows the same structure aside from adaptations specific to channel tables. Its implementation is presented in Figure 4.3 for comparison.

[3] Semantically speaking, "higher" priority processes are assigned lower priority numbers. See Section 3.1.5 for more details about how they are computed.

```
local ctx = {                                   -- Global state maintained by ssm-lua
    running_process = nil,                      -- Current running process
    current_time = 0,                           -- Current logical timestamp
    event_queue = Heap("earliest"),             -- Priority queue of scheduled variables
    run_queue = Heap("prio"),                   -- Priority queue of processes
}

local function channel_do_update(c)
    local next_earliest = math.huge
    for k, v in pairs(c.later_value) do
        local t = c.later_time[k]
        if t == ctx.current_time then           -- This update is scheduled for now
            c.value[k] = v
            c.last_updated[k] = t
            c.later_value[k] = nil
            c.later_time[k] = nil
        else                                    -- This update is scheduled for later
            next_earliest = math.min(next_earliest, t)
        end
    end

    c.earliest = next_earliest
    if c.earliest ~= math.huge then
        enqueue(ctx.event_queue, c)             -- There are still pending updates
    end

    for p, _ in pairs(c.triggers) do
        if p.scheduled then
            enqueue(ctx.run_queue, p)           -- Schedule sensitive processes
        end
    end
    c.triggers = {}
end

function ssm.tick()
    ctx.current_time = next_scheduled_time(ctx.event_queue)
    for c in scheduled_events(ctx.event_queue, ctx.current_time) do
        channel_do_update(c)
    end
    for p in scheduled_processes(ctx.run_queue) do
        process_resume(p)
    end
end
```

Figure 4.3: ssm-lua's `tick` function, which closely follows the structure of the SSM runtime's tick function (Figure 4.2). The `channel_do_update` function performs each scheduled update and parallels the Update helper from Figure 4.2.

```
globals
  ▹ EventQueue    // Priority queue of scheduled variables, ordered by later_time

procedure TickLoop [no I/O]                  // The main event loop
    call Tick                                // Run the program for the initial instant (time zero)
    forever
        wait until next event in EventQueue  // Pause until next event (unnecessary for simulation)
        call Tick                            // Run the instant at time of next delayed assignment
```

Figure 4.4: Running an SSM system without I/O or under simulation is simple: just call Tick repeatedly.

4.2 Input and Output Variables

Executing an SSM system is simple if it does not need to interact with its environment to perform I/O. We can run the system "under simulation" by repeatedly calling the `tick` function from Figure 4.2 until the event queue is empty. This execution strategy does not synchronize the system with physical time, and is ideal for testing (e.g., Section 3.2.3) because we do not need to wait for logical delays to elapse. We can also add (physical time) pauses to the simulation tick loop, as shown in Figure 4.4, to have it reflect the logical timing behavior prescribed by the program.

To support I/O, our runtime relies on a platform-specific driver to relay inputs and outputs to and from the SSM program. These events are timestamped using a platform-provided timer and conveyed through scheduled variables: *input variables* are assigned when the system receives an external input event, while *output variables* actuate some external effect when assigned by the program. Otherwise, variables behave the same as regular scheduled variables (those allocated by **ref**), and their implementation is abstracted from the SSM scheduler and program.

Figure 4.5 shows an idealized tick loop that illustrates how an SSM driver manages input and output events, for one input variable and one output variable. Between ticks, input events may interrupt the suspended system and cause assignments to the input variable to be added to the event queue; these are immediately processed by tick function. Afterwards, if any assignment was made to the output variable, the driver forwards that event to the environment by emitting the assigned value.

```
globals
  ▹ EventQueue   // Priority queue of scheduled variables, ordered by later_time
  ▹ InputVar     // Scheduled variable reflecting external inputs
  ▹ OutputVar    // Scheduled variable conveying external outputs

procedure TickLoop [idealized]
    call Tick
    forever
        wait until next event in EventQueue or input event   // Suspend the system
        if there was input from environment                   // Did the environment deliver input?
            enqueue InputVar in EventQueue                    // ... yes: relay input to scheduler
        call Tick                                             // Advance SSM logical time
        if OutputVar was updated                              // Did the program produce output?
            emit output to environment                        // ... yes: relay output to environment
```

Figure 4.5: A tick loop that idealizes many aspects of performing timed I/O, for purposes of illustration. This pseudo-code assumes that input events only arrive while the system is suspended at the **wait** statement.

Ironically, this idealized tick loop assumes that execution is synchronous. It relies on input events only ever appearing while the system is suspended at the **wait** statement, so it requires that all other statements execute instantaneously. In doing so, it eludes two challenges that a realistic tick loop must contend with. First, inputs may arrive at any point: asynchronously updating the input variable while the tick function is running may corrupt system state. Second, the tick loop must ensure that logical time advances monotonically: it should only tick to an instant once the corresponding physical time has passed, to account for any possible input events prior to that time. A reasonable algorithm should also tolerate bursty workloads where logical time is temporarily unable to keep up with physical time.

Our real driver implementation, which runs on microcontroller hardware, uses an RTOS-provided semaphore to suspend and wake the system, and adds an *input queue*—distinct from the scheduler's event and run queues—to manage input events from the environment. The input queue is a first-in first-out ring buffer large enough to accommodate a modest backlog of input events before having to drop new ones. Figure 4.6 illustrates this input handling architecture. Our driver's tick loop forwards events from the input queue to the event queue, and calls the tick function to let the system react to those inputs. It sleeps by waiting on the semaphore, with an

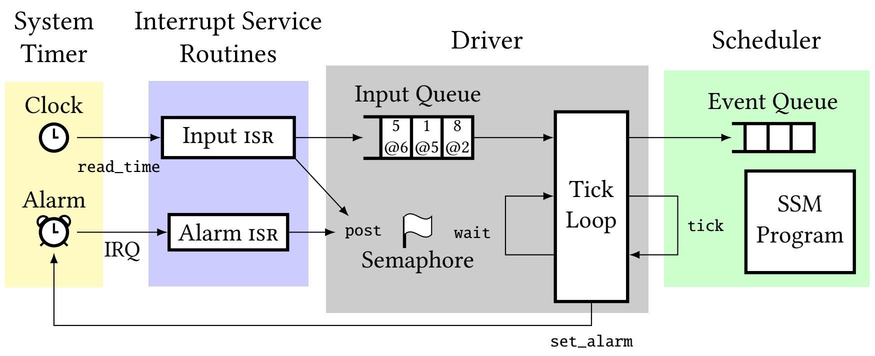


Figure 4.6: SSM system block diagram, with input events timestamped in software, inside the input ISR, using the system clock. The tick loop suspends the system by waiting on a semaphore, and is awoken when either the input or alarm ISR posts to that semaphore. Output events are also handled in software, by a low-priority process (not shown).

alarm scheduled to post to the semaphore when the sleep period is over. Meanwhile, external inputs trigger an *interrupt service routine* (ISR) that adds a timestamped event to the input queue and posts to the semaphore to wake up the (potentially sleeping) system.

Keeping the input queue separate from the scheduler's event queue avoids excessive synchronization costs. The input queue is loaded asynchronously by ISRs and emptied by the tick loop, but it is a simple ring buffer that is easy to make thread-safe. By contrast, the scheduler's event queue is a priority queue, to which events are added and sometimes removed out-of-order, by both the tick loop and running SSM processes. However, because these run synchronously, the event queue is never accessed asynchronously and does not need to be made thread-safe.

Figure 4.7 shows the pseudocode for our microcontroller tick loop. At each iteration of the tick loop, the runtime checks the input queue for events to schedule in the event queue. Events in the input queue will be in increasing order since we assume system time advances monotonically. Pending inputs are forwarded to the event queue if and only if they occur before the earliest in the event queue. Otherwise, or if there is no pending input event, the runtime executes the SSM program by calling the tick function, advancing logical time. Finally, if there are neither input events to process nor internal events (delayed assignments) ready to execute, the tick loop goes to sleep, blocking on a semaphore until either its alarm expires or some fresh input appears. Upon waking, the tick loop cancels the alarm and resets the semaphore to prevent stale posts on the

```
globals
  ▹ EventQueue    // Priority queue of scheduled variables, ordered by later_time
  ▹ InputQueue    // FIFO queue of input events; populated by input ISR, which posts to Semaphore
  ▹ InputVar      // Scheduled variable reflecting external inputs
  ▹ OutputVar     // Scheduled variable conveying external outputs
  ▹ Semaphore     // Used to suspend and wake the system
  ▹ Clock         // Used to read system time
  ▹ Alarm         // Used to trigger alarm ISR, which posts to Semaphore

procedure TickLoop
    call Tick
    forever
        let pt ← read Clock                       // Read physical time
        let lt ← earliest in EventQueue           // Logical time of next scheduled event
        if InputQueue has event earlier than lt   // Pending input event before any other event?
            enqueue InputVar in EventQueue        // ... yes: safe to add to event queue
        else if lt earlier than pt                // Has logical time fallen behind physical time?
            call Tick                             // ... yes: run for an instant, advnacing logical time
        else if lt = ∞                            // Is there an event scheduled for the future?
            wait on Semaphore                     // ... no: sleep until woken by input event
        else
            set Alarm to lt                       // ... yes: schedule an alarm to wake up
            wait on Semaphore                     // Sleep until alarm or input event
            cancel Alarm                          // If an input event woke us, cancel the alarm
            reset Semaphore                       // Reset semaphore, in case alarm raced with input
```

Figure 4.7: The tick loop from the microcontroller platform driver.

semaphore lingering into the following loop iterations.

Note that the tick loop always reads the physical time before checking the input queue. Otherwise, it is possible for an input event to occur after checking the input queue, but before reading the system clock, causing logical time to advance past the input event. When that input is processed in the next loop iteration, logical time will run backwards and violate monotonicity.

4.3 Timestamp Peripherals

Earlier implementations of SSM [53, 63] managed I/O timestamps in software. The microcontroller is configured to directly trigger an ISR upon receiving input. This ISR, shown in Figure 4.8, records the current physical time as quickly as possible, then attempts to enqueue an event with that timestamp and value from the peripheral in the input queue. It uses an hardware-provided

```
globals
  ▹ InputPeripheral    // Where inputs come from
  ▹ InputQueue         // FIFO queue of input events
  ▹ Semaphore          // Used to suspend and wake the system
  ▹ Clock              // Used to read system time
  ▹ IRQLock            // Used to prevent nested interrupts

procedure InputISR
    acquire IRQLock                     // Ensure inputs are enqueued in timestamp order
    let $t$ = read Clock                // Read physical time as soon as possible
    let $v$ = read InputPeripheral      // Read value from input device
    if InputQueue is not full           // Check capacity; excess input events are dropped
        enqueue (v@t) to InputQueue     // Add event to input queue
    release IRQLock                     // Done with input queue
    post to Semaphore                   // Wake up the (potentially sleeping) tick loop
```

Figure 4.8: An ISR that timestamps input events in software.

IRQ lock to ensure enqueued input events appear with non-decreasing timestamps, at the cost of temporarily disabling nested interrupts. Output events are handled by low-priority SSM processes which simply **wait** for assignments and forward them to the appropriate peripheral.

This software-based approach is flexible and portable (it only uses widely available platform features like semaphores and alarms), but it is imprecise. Input timestamps are affected by interrupt response time uncertainty, while output latency is dependent on the execution time of that instant.

To address these issues, our SSM implementation also supports using *timestamp peripherals* as an interface between hardware ports and the software runtime. Timestamp peripherals run alongside the CPU and offload its timed I/O duties: they timestamp input events before delivering them to software, and emit output events according to software-specified timestamps.

Timestamp peripherals are conceptually simple, but they require hardware support and are not readily available. They must be implemented using a cycle-accurate co-processor, or directly in hardware, to achieve timing precision. While a handful of existing devices can timestamp I/O events, most are specialized, expensive, and hard to obtain. Implementing custom timestamp hardware in an FPGA is less accessible still.

Instead, we implemented SSM on the RP2040 microcontroller, which is inexpensive (US$0.70)

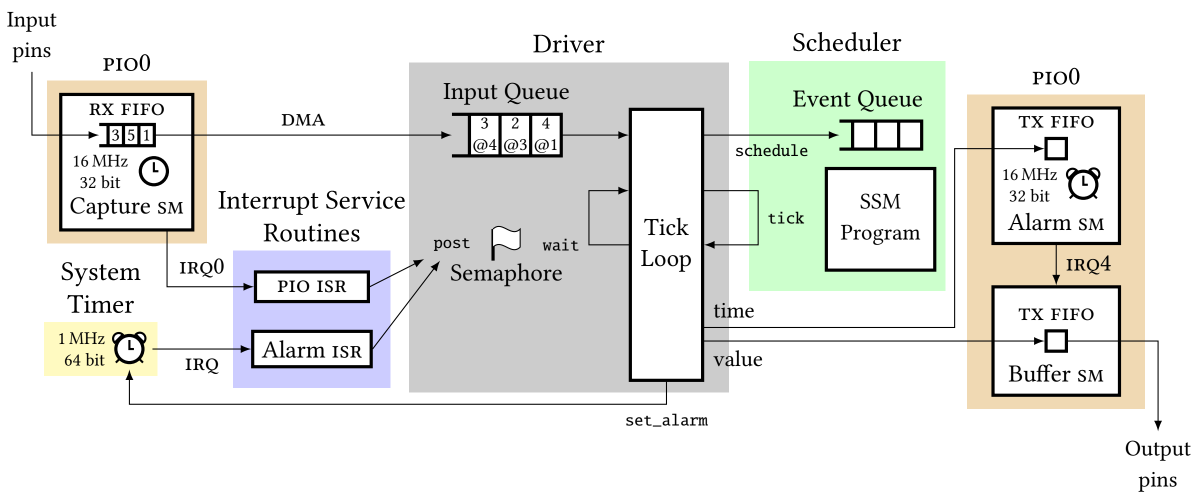


Figure 4.9: SSM system block diagram on the RP2040. The capture SM (in PIO0) timestamps input pin events; the DMA controller enqueues them. The tick loop (Figure 4.7) gathers the next event from the input queues, schedules it in the SSM event queue, calls `tick` to run the SSM program for an instant, feeds an updated event to the output SMs (also in PIO0), sets an alarm with the system timer, and sleeps.

and widely available (at this time of writing). Our implementation builds timestamp peripherals using the RP2040's Programmable I/O (PIO) hardware. Although the PIO was not designed for implementing timestamp peripherals, it is fast, predictable, and powerful enough to do so. These cycle-accurate devices are isolated from main processor delays, and let us manage I/O at the same precision as the rest of the microcontroller hardware (which uses an external crystal oscillator).

The RP2040 provides two PIO blocks which execute tiny PIO-specific assembly language programs, and are designed as conduits between the RP2040's 30 GPIO pins and its ARM cores. Each PIO block comprises four independent state machines (SMs) that share 32 instructions of program memory and execute instructions in lockstep. We use one SM to implement an *input capture* device, which maintains a counter in a precisely timed loop and emits timestamped input update events when it sees a change in input levels.

Meanwhile, we use two SMs to implement an *output compare* system that consists an alarm and a buffer. The alarm SM also runs in a precisely timed loop where it checks a counter against an alarm time. When the counter reaches the alarm time, the alarm SM signals to the buffer SM to emit stored value of the output event. In fact, our uses two strategies to transmit output variable assignments to the environment: for sufficiently delayed assignments, the output system

```
void start_pio_counters(void)
{
    uint32_t tmr = timer_hw->timerawl;    // Read low bits of 1 MHz system timer
    uint32_t ctr = ~((tmr + 3) << 4);     // Convert to 16 MHz countdown value

    pio_sm_put(pio0, capture_sm, ctr);    // Send initial count to input SM
    pio_sm_put(pio0, alarm_sm, ctr);      // ... and output SM
    pio_set_sm_mask_enabled(pio0,         // Start SMs in sync
            CAPTURE_SM | ALARM_SM | BUFFER_SM, true);
}

time_t read_timer(void)
{
    uint32_t lo = timer_hw->timelr;       // Read low bits (and latch high bits)
    uint32_t hi = timer_hw->timehr;       // Read high bits
    uint64_t us = ((uint64_t) hi << 32) | lo;
    return us << 4;                       // Convert 1 MHz timer to 16 MHz
}
```

Figure 4.10: Using the RP2040's 1 MHz system timer to initialize the PIO SM counters, and to read 16 MHz-resolution timestamps used by SSM. Reading the lower 32 bits from the system timer's `timelr` register latches the upper 32 bits in the `timehr` register, to avoid a wraparound race.

sets the alarm SM's target counter to trigger the buffer SM when the event is scheduled for. For instantaneous assignments, shorter delays, and when the system is running behind, the output system instructs the buffer SM to immediately emit the event, rather than risk missing the output deadline while programming the alarm SM. The result is high-precision output timing when possible, and best-effort timing otherwise.

These timestamp peripherals communicate with the rest of the system according to the block diagram in Figure 4.9. Note that although the RP2040 provides two PIO blocks (each with four SMs), our driver only uses three SMs from a single PIO block (one SM for input; two for output), to avoid the complexity of synchronizing between independent PIO blocks. The driver's tick loop follows the pseudocode shown in Figure 4.7, though it additionally forwards output variable assignments to the output system upon returning from the tick function.

4.3.1 Timestamps and Clock Synchronization

Our RP2040 platform driver uses 64-bit timestamps (`time_t`) that count at 16 MHz (62.5 ns), which we chose due to our PIO programs running 8-cycle loops at 128 MHz and the RP2040 system timer running at 1 MHz. At this speed, 32-bit timestamps would wrap around in under 5 min; 64-bit timestamps give us 36,533 years.[4]

We use the RP2040's 1 MHz system timer as the master clock, which measures time since it was started. Our SSM runtime's 16 MHz clock is a power-of-two multiple of the system timer frequency, which lets us efficiently convert between the time bases with bit-shifting (Figure 4.10). Because the PIO programs cannot directly read the system timer, we maintain two additional real-time clocks in the PIO programs that need access to the current time. All three timers are driven by clocks derived from the external 12 MHz crystal; so they will remain synchronized if we set them to run at precisely the same rate and in phase.

We initialize the PIO counters with the code in Figure 4.10, which reads the system timer, sends the initial count value to the counting SMs, and starts all the three SMs simultaneously. The initialization routine compensates for its own latency, which we experimentally determined to be roughly 3 μs. We add this offset to the initial PIO counter to ensure it runs slightly *ahead* of the system clock. This offset is critical for the correctness of the tick loop, which assumes that if the PIO input queue is empty, future queued events will have a greater timestamp than the current system clock time. If the PIO counters were run behind the system clock, PIO timestamps could be smaller, violating this assumption. We ensure our clocks are synchronized using the loopback test described in Section 4.4.

4.3.2 The Input System

The input system uses a single PIO SM to sample a group of input pins at 16 MHz and send a sequence of timestamped changes to the platform runtime. This capture SM is conceptually simple: it reads an initial counter value from the CPU to synchronize with the system timer, then enters a loop that increments the counter and polls the input pins. If any input pin state has changed, the capture SM emits the new pin values and current counter value into a FIFO, and interrupts the CPU to notify it of the input event.

[4] Comfortably outlasting the span of my research career.

```
; Registers:
;         x      the previous GPIO pins' values
;         y      counter value (decreasing)
;         isr    for reading GPIO pins
;         osr    scratch register (used to back up y because only x and y can be compared)
.program input_capture
    pull                                  ; Read initial counter value to osr
    mov  y, osr                           ; ... and save it to y
    in   pins, INPUT_PINCOUNT             ; Read initial GPIO pin state to isr
    in   null, 32 - INPUT_PINCOUNT        ; ... and pad unused pin bits
    mov  x, isr [13]                      ; Save initial GPIO pin state to x
                                          ; ... and stall 13 cycles to align with output SMs

start:                                    ; Start of loop
.wrap_target                              ; Alternative "label" for start
    jmp  y--, next                        ; Decrement counter (and "jump" to next)
    next:
    mov  osr, y                           ; Back up counter value (need y for comparison)
    in   pins, INPUT_PINCOUNT             ; Read initial GPIO pin state to isr
    in   null, 32 - INPUT_PINCOUNT        ; ... and pad unused pin bits
    mov  y, isr                           ; (Only x and y can be compared on PIO)
    jmp  x!=y, changed                    ; Jump if state changed
    mov  y, osr                           ; Restore counter value
    jmp  start                            ; Restart the loop

changed:                                  ; Input value change: send event
    mov  x, isr                           ; Remember new value
    push noblock                          ; ... and enqueue it to the RX FIFO
    mov  isr, osr                         ; Read the current counter value
    push noblock                          ; ... and enqueue it to the RX FIFO
    irq  0                                ; Notify CPU of the event (see below)
    mov  y, osr                           ; Restore counter value
    jmp  y--, start [3]                   ; Decrement counter
                                          ; ... and stall 3 cycles to stay in 16-cycle phase
.wrap                                     ; Restart the loop (wraps to .wrap_target)
```

```
void pio_irq0_isr(void)                  // Triggered by irq 0
{
    sem_post(&sem);                      // Post to semaphore; awaken tick loop
}
```

Figure 4.11: The input capture PIO program. Every 8 cycles, the state machine checks the input pins and, if they have changed, pushes the new value and the current counter value to the RX FIFO. Executing `irq 0` executes an ISR that posts to the semaphore to awaken the tick loop.

```
time_t pio_to_time(uint32_t ctr)
{
    time_t himask = ~(((time_t) 1 << 32) - 1);
    time_t rt = read_timer();                // Read system timer (see Figure 4.10)
    time_t hi = rt & himask;                 // ... and get its high bits
    time_t lo = (time_t) ~ctr;               // Invert PIO's decrementing counters
    if (lo & (1 << 31) && !(rt & (1 << 31)))
        hi -= (time_t) 1 << 32;              // Correct near 32-bit epoch
    return hi | lo;                          // Combine low and high bits
}

uint32_t time_to_pio(time_t t)
{
    return ~(uint32_t) t;                    // Invert lower 32 bits of SSM time
}
```

Figure 4.12: Conversion between 64-bit SSM time and the PIO's 32-bit 16 MHz decrementing counters. We take the top 32 bits from the system timer, being cautious around 32-bit epochs.

The actual PIO code for this (Figure 4.11) is complicated because the PIO instruction set is highly idiosyncratic. For example, only the two scratch registers `x` and `y` can be compared, and decrement can only be done as part of a conditional jump. To compensate for this limitation, we complement PIO counter values when we convert them to and from the system timestamps that SSM uses (Figure 4.12).

We have carefully written our PIO code so that the counter decrements every 8 cycles regardless of any input change, keeping the counter synchronized with the system timer. We run the PIO at 128 MHz, so our code samples and timestamps inputs at 16 MHz. However, our implementation cannot resolve *consecutive* events occurring faster than 8 MHz: when the capture SM detects an input event, it takes extra instructions to send the captured event to the CPU. We pad these instructions to 8 cycles to keep the counter decrementing at a constant rate.

To allow our system to handle longer input event bursts, we program a channel of the RP2040's DMA controller to empty the 4-word hardware RX FIFO from the capture SM into a 64-word ring buffer in main memory. We leverage the controller's built-in support for power-of-two-sized ring buffers, and use a trick to make the transfer continue indefinitely: a second channel, configured to start the moment the first channel completes, restarts the first channel.

```
; Registers:
;         x      the alarm target value
;         y      counter/timestamp value (decreasing)
;         isr    unused
;         osr    reads TX FIFO for new alarm target
.program output_alarm
    pull noblock          ; Read initial counter
    mov  y, osr
again:                    ; To stay in 8-cycle phase
    nop                   ; ...stall for 1 cycle
.wrap_target
    jmp  y--, dec [3]     ; Decrement counter
                          ; ...and stall 3 cycles
dec:
    pull noblock          ; Poll alarm target
    mov  x, osr           ; Prepare for comparison
    jmp  x!=y, again      ; Loop if target not reached
    irq  4                ; Interrupt output buffer
.wrap                     ; Loop to .wrap_target
```

```
; Registers:
;         x      unused
;         y      unused
;         isr    reads the initial GPIO state
;         osr    holds buffered output
.program output_buffer
    in pins, 32       ; Initial GPIO state
    mov osr, isr      ; ...as default output
.wrap_target
    wait 1 irq 4      ; Wait for alarm SM
    out pins, 32      ; Write osr to GPIO
.wrap                 ; Loop again
```

Figure 4.13: The output alarm and buffer PIO programs. Every 8 cycles, the alarm SM decrements its counter, reads a new alarm target if the CPU has written one, and sends an interrupt to the output SM if the counter matches the target. Alarm counters are placed in the TX FIFO and read into the `osr` register using `pull noblock`; when the TX FIFO is empty, `pull noblock` reads from `x` instead, which is a nop in this program. The buffer SM holds a value from the CPU (Figure 4.14); it outputs this value when it receives `irq 4` from the alarm SM.

4.3.3 The Output System

Our runtime's output system allows the SSM program to schedule a single new value to be placed on the output pins at a specific 16 MHz timestamp in the future. This is achieved by using two PIO SMs: the *alarm SM* acts as a real-time alarm that triggers the *buffer SM* to emit an updated value on the pins at the scheduled time. This split is necessary because an SSM program can "change its mind" about when and which outputs need to be emitted. SSM semantics allow only one pending event per variable, but allows that pending event to be overwritten, which is useful when, say, handling timeout behavior. As such, we needed the output system to be able to reschedule an alarm and the value to be written at that time, and the PIO's compulsory per-SM FIFOs were getting in the way.

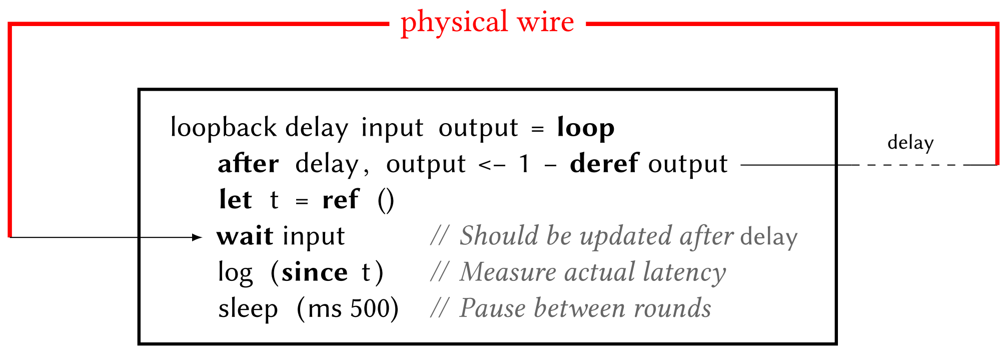
Figure 4.13 shows the PIO code for our output SMs. After reading an initial counter value to synchronize with the system timer, the alarm SM enters an 8-cycle loop, which we padded

```
#define OUTPUT_MARGIN   32      // Don't use alarm SM when deadline is <2 µs away

void sched_pio_out(time_t t, uint32_t v)
{
    // Enqueue new buffer output value in TX FIFO
    pio_sm_put(pio0, buffer_sm, v);

    // Make the buffer SM read this output value: inject a pull instruction
    pio_sm_exec(pio0, buffer_sm, pio_encode_pull(0, 1));

    if (read_timer() + OUTPUT_MARGIN <= t) {
        // Enough time left to set up output alarm
        uint32_t tgt = time_to_pio(t);      // Convert to PIO counter (Figure 4.12)
        pio_sm_put(pio0, alarm_sm, tgt);    // Set alarm SM target
    } else {
        // Deadline too close: emit the output value by injecting an out instruction
        pio_sm_exec(pio0, buffer_sm, pio_encode_out(pio_pins, 32));
    }
}
```

Figure 4.14: Scheduling an output event with the alarm and buffer SMs. If the time to the scheduled output deadline falls below the `OUTPUT_MARGIN`, this function bypasses the alarm SM and instructs the buffer SM to immediately emit the output value.

with stalling to operate at the same frequency as the input capture loop. Each loop iteration, the alarm SM checks for an updated alarm target before decrementing the counter and comparing it against the current alarm target, sending an interrupt to the buffer SM when the alarm target matches the counter. The buffer SM (Figure 4.13) program blocks on IRQ4 before sending data to the output pins (IRQ4 is only visible within the PIO block where the alarm and buffer SMs reside, and used for synchronizing SMs).

The main processor changes the alarm target and the buffer data by writing to their respective FIFOs, as shown in Figure 4.14. Because the buffer SM does not poll its FIFO like the alarm SM, the main processor injects a `pull` instruction to force the buffer SM to read the new data from its FIFO. If there is not enough time to set up an alarm-triggered output, it injects an `out` instruction to directly emit the output at the expense of precise timing. This happens when an SSM program makes an instantaneous assignment to the output variable, or when it schedules a delayed assignment for too soon.



```
loopback delay input output = loop
    after delay, output <- 1 - deref output
    let t = ref ()
    wait input        // Should be updated after delay
    log (since t)     // Measure actual latency
    sleep (ms 500)    // Pause between rounds
```

Figure 4.15: A loopback program, tested with the input and output pins connected externally.

4.4 The Loopback Test

To validate the physical timing of our implementation against the logical behavior prescribed by SSM, we test our system with the SSLang program in Figure 4.15. This program schedules a delayed assignment to the output pin, and awaits events on the input pin. Externally, we connect the output pin to the input pin, so assigning to output indirectly updates input via that physical loopback connection.

When implemented correctly, input and output variables should not be distinguishable from regular scheduled variables, so the loopback function should behave the same whether input and output refer to shorted external I/O pins or the same internal scheduled variable, i.e.:

```
let x = ref 0          // Internal scheduled variable
loopback delay x x     // Both input and output refer to x
```

The scheduled variable t helps us identify any discrepancy between the two scenarios. The program allocates t at the same (logical) time as when the delayed assignment to output is scheduled, so after the **wait** statement unblocks **since** t measures the time elapsed until the assignment to input. In particular, **since** t is equal to delay when input and output are internally connected (i.e., loopback delay x x), so they should also be equal when input and output are externally connected via a wire as illustrated in Figure 4.15.

This loopback test is very sensitive to clock synchronization issues, so we used it to calibrate our runtime's various clocks. In earlier runs of this experiment, we observed that the measured delay consistently lagged 1 tick behind the prescribed delay. This error arises when the input and output SMs' synchronous loops are not correctly phase-aligned, leading the input SM to sample the GPIO pin before the output SM writes the pin. We fixed this lag by starting the input SM a few instructions later to put it in phase (see Figure 4.11).

We also used this loopback test to determine the 3 μs offset applied during the SM initialization procedure (see Section 4.3.1). Because the delayed assignment to output takes place in the same instant as the event on input, a poorly calibrated system clock—running ahead of the PIO timers—would lead the tick loop to execute the instant before waiting long enough for the input event. When the tick loop tries to schedule the "late"-arriving input event in a later iteration, the SSM runtime complains that it cannot schedule a delayed assignment for an instant it has already executed, and throws a runtime error.

With our runtime correctly calibrated, the loopback program consistently measures the same physical delay (**since** t) as the logically prescribed delay, provided that delay is long enough. When delay is less than 17 μs, the measured duration becomes unpredictable because the output system is unable to accurately respond in so little time. This threshold of 17 μs is consistent with our findings from the blinker experiment, discussed later in Section 5.1.

Chapter 5: Evaluation

SSM prescribes a program's temporal behavior, but its abstractions are built upon the synchronous fiction that non-blocking computation is logically instantaneous. In reality, those abstractions are implemented using priority queues, stackful coroutines, and heap-allocated objects, which come at a non-zero physical time cost.

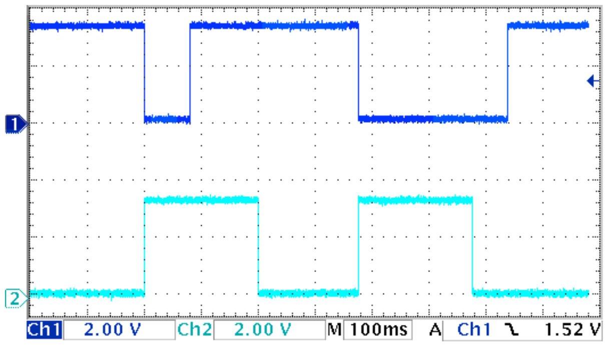
In spite of that overhead, our SSM implementation is precise and predictable in real-world applications. For instance, Figure 5.1 shows a SSLang program, buttonpulse, which emits a 200 ms pulse when triggered by a bouncy push-button switch, as illustrated by the oscilloscope traces in Figure 5.2. Aside from its 20 μs reaction time, the system reliably produces the behavior specified by the program.

This chapter discusses several experiments that evaluate our SSM implementation, focusing on SSLang and its runtime on the RP2040 platform [87].[1] Sections 5.1 and 5.2 explores the limits of the output and input systems, respectively; Section 5.3 investigates the reliability of the system under high input load; Section 5.4 compares a SSLang program's latency and timing accuracy to a C equivalent.

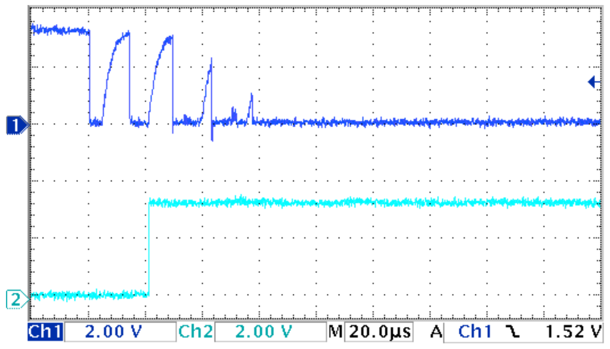
[1] We had previously designed and conducted several similar experiments with our collaborators Robert Krook, Bo Joel Svensson, and Koen Claessen [52, 63]. Those experiments were performed using an earlier version of our SSM runtime, which used the Zephyr real-time operating system [86] and ran on a Nordic Semiconductor NRF52840-DK board [78]. Kent J. Hall helped us integrate our build system with Zephyr's.

```
sleep d =            // Pause for duration d         waitfor x v =         // Pause until variable x
    let x = ref ()                                       while deref x != v     // ... has value v
    after d, x <- ()                                         wait x
    wait x

debounce delay input press = loop                    pulse period press output = loop
    waitfor input 0    // Button pressed                 wait press        // Wait for "press" event
    press <- ()        // Send "press" event             output <- 1       // Pulse high immediately
    sleep delay        // Debounce                       after period,     // ... and low later
    waitfor input 1    // Button released                    output <- 0
    sleep delay        // Debounce                       wait output       // Wait for end of pulse

buttonpulse button led =
    let press = ref ()                                   // This debounced button press signal
    par debounce (ms 10)   button press                  // ... coordinates the debouncer
        pulse    (ms 200)  press  led                    // ... and the pulse program
```

Figure 5.1: A debouncing pulse generator in SSLang. The debounce function converts bouncy, active-low push-button inputs into pure press events, which trigger the pulse function to emit a 200 ms pulse. The program's entry point, buttonpulse, runs debounce and pulse in parallel.



(a)



(b)

Figure 5.2: Behavior of the debouncing pulse generator from Figure 5.1, running on our RP2040 platform SSM runtime. In each waveform, the top trace (blue) is the active-low push-button input; the bottom (cyan) is the LED output. (a) The program emits two 200 ms pulses in response to two button presses. (b) Zoomed in, button bounces and the 20 µs reaction time become visible.

```
blinker led = loop
    val <- deref led
    let toggled = 1 - val
    after us 13, led <- toggled
    wait led
```

Figure 5.3: The fastest blinker on the RP2040 platform, which generates a stable 38.46 kHz square wave.

```
pulsewidth input =
  let pw = ref 0          // Stores pulse width
  par loop                // Measure pulse width
        wait input        // Rising edge event
        let b = ref ()    // Take note of the time
        wait input        // Falling edge event
        pw <- since b     // How long it has been
      loop                // Log pulse width
        sleep (ms 1000)   // Pause between logging
        log (deref pw)    // Report pulse width
```

Figure 5.4: A program that measures pulse width, which it periodically logs to the console in a parallel process. The implementation of the log function is omitted for brevity.

5.1 Blinker

The SSM runtime is designed to reflect the timing prescriptions of our programming model, so its timestamp peripherals prioritize precision and accuracy at the cost of some latency. Thus, SSLang encourages programmers to interact with output peripherals via delayed assignments to compensate for the latency of its output system. Longer delays mean the output is scheduled far enough in advance for the system to make use of the cycle-accurate alarm SM to time the output.

To explore the limits of this output system, we incrementally adjusted a high-frequency blinker program that toggles the value of an output pin (Figure 5.3). We discovered that our RP2040 runtime implementation is able to achieve timely output in as little as 13 µs. Its performance is limited by a combination of factors. Some are inherent to the RP2040 platform: when the system alarm goes off to indicate a delayed assignment event, the system experiences some interrupt latency before the alarm ISR can post to our runtime's semaphore, and further dispatch latency while the RTOS context switches to the main loop blocking on that semaphore. Other factors stem from our SSM runtime: its tick loop (Figure 4.7) must check the (empty) input queue and run the tick function at each iteration before forwarding the delayed assignment to the output system (Section 4.3.3).

The blinker program's timing accuracy suffers greatly when delayed assignments are scheduled less than 13 µs away. This happens because when the output deadline falls within the 2 µs

Pulse Input	Expected	Observed	Jitter	Error
80 ms	1 280 000	1 280 021	1	22
8 ms	128 000	128 002	1	3
800 µs	12 800	12 800	1	1
80 µs	1 280	1 280	1	1
8 µs	128	128	1	1
800 ns	12.8	13	1	0.2
80 ns	1.28	2		0.72
40 ns	0.64	2		1.36

Figure 5.5: Measurements by the SSLang pulse timer from Figure 5.4. Units are 16 MHz clock cycles (62.5 ns). Jitter is not recorded for pulses of less than 80 ns because error is too significant.

margin, the output system to bypasses the alarm SM and emits events on a best-effort basis (Figure 4.14). We could not reduce this margin any further without risking the alarm SM missing output deadlines entirely.

5.2 Pulse Timer

Our SSM runtime implementation for the RP2040 uses timestamp peripherals to capture input events in hardware. Doing so allows high-level SSLang programs to measure and produce signals with 62.5 ns precision, far more accurately than is possible using only the RP2040's 1 MHz system timer. To demonstrate the precision and limits of SSLang's input timestamping, we implemented a program that measure the width of input pulses (Figure 5.4).

We tested our pulse timer with periodic pulses of varying widths at 10 kHz, and recorded the difference in timestamps between pulse edges; Figure 5.5 shows our results. We observed a single least-significant bit of jitter in all cases, likely an artifact of sampling. We attribute the roughly 20 ppm errors in the long-period measurements to the expected precision of the crystal oscillator. And while we do not expect correct results for pulses shorter than the 62.5 ns sampling period, we were pleased that the resulting behavior was not absurd. The input SM was still able to observe certain pulses and conclude that they were short.

When we conducted the same experiment with pulse signals generated at frequencies greater than 200 kHz, we observed occasional but drastic measurement errors. For instance, with a

```
freqcount input =
    let count = ref 0                    // Counts the number of input events
        clock = ref ()                   // Used for periodic logging
    after ms 1000, clock <- ()
    loop
        wait clock || input              // Block until either is assigned
        if since clock != 0              // Did we unblock because 1 s has passed?
            count <- deref count + 1     // ... no: count input event that unblocked us
        else
            log (deref count)            // ... yes: time to log current input count
            after ms 1000, clock <- ()   // Pause for 1 s
            wait gate                    // ...
            after ms 1000, clock <- ()   // ... and resume counting
            if since input == 0          // Was input assigned just now?
                count <- 1               // ... yes: reinitialize count to 1
            else
                count <- 0               // ... no: reinitialize count to 0
```

Figure 5.6: A frequency counter program, written in SSLang. It counts and reports the number of events on an input variable each second.

320 kHz pulse signal with a 500 ns pulse, the program occasionally reports 808 or 809 ticks instead of the expected 8 or 9. These errors are due to incoming input events accumulating faster than the program can process them, overflowing the input ring buffer. For our implementation, it takes 32 events—16 cycles of the pulse signal—to overflow a 256 B ring buffer; at 320 kHz, 16 cycles is 50 μs, accounting for the extra 800 ticks we observe.

5.3 Frequency Counter

Our pulse timer demonstrated that our input system can be overwhelmed by a deluge of input events. After all, SSM was designed for sparse workloads rather than for throughput, and cannot handle such excessively eventful environments. However, the pulse timer's failures were sporadic and difficult to quantify. The program only measures the duration between adjacent input events, so its behavior is not affected when only a few events are dropped from the overflowing input queue.

To better understand how our runtime behaves under high input load, we use SSLang to implement a frequency counter, whose code is shown in Figure 5.6. This program measures the

Frequency	Expected Events	Observed Events
30 kHz	60000	60000
40 kHz	80000	74271
50 kHz	100000	72670
60 kHz	120000	71390
70 kHz	140000	70013
80 kHz	160000	68574
>90 kHz	180000	unstable

Figure 5.7: Events observed by the frequency counter program in Figure 5.6.

frequency of an input signal. Unlike the pulse timer, the frequency counter's behavior is affected by any dropped input event, because it must count every event that appears on its input variable in order to determine their frequency.

We test our frequency counter with a square wave signal, which produces twice the number of events as the frequency of the signal, one each for the rising and falling edges. We are able to reliably measure frequencies below 37 kHz (74000 events), with only 1 Hz of error due to sampling artifacts. Beyond this frequency, the program remains responsive, though it consistently logs lower event counts than expected. For instance, at 50 kHz, the program reports 71390 ± 1 events instead of the expected 100000. Figure 5.7 records these observations.

Above 90 kHz, the frequency counter's event counts are no longer stable, though they continue to decrease as we push the input frequency higher. The program continues to respond and report (wildly inaccurate) frequency counts up to 740 kHz. Beyond that, the program freezes: the processor spends all of its cycles thrashing within the PIO ISR, preventing the tick loop from making any progress.

As its name suggests, SSM works best with sparse events, but our implement also tolerates input bursts. This capability is important in real-world applications that must accommodate bouncy switches like buttonpulse (Figure 5.1). We found that the frequency counter can also successfully handle 3 MHz bursts of 28 events with no loss of accuracy; the DMA controller could move them all to the 256 B ring buffer before the software had to start processing them. A larger buffer would handle longer bursts.

```
// Triggered by rising edge of the input pin
void gpio_rise_isr(void)
{
    gpio_put(OUTPUT_PIN, 1);
    busy_wait_us(100);
    gpio_put(OUTPUT_PIN, 0);
}
```

(a) Low-level C implementation, which polls the system timer in an interrupt handler. Additional code (not shown) configures the GPIO input pin to trigger this ISR.

```
pulse input output = loop
    wait input
    inval <- deref input
    if inval == 1
        output <- 1
        after (us 100), output <- 0
```

(b) SSLang implementation, which uses an instantaneous assignment for the rising edge of the output, and a delayed assignment for the falling edge.

Figure 5.8: Two implementations of a reactive 100 µs pulse generator.

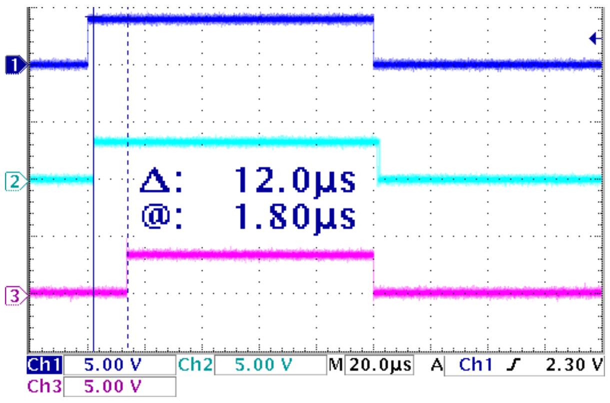
5.4 Reactive Pulse Generator

To evaluate our SSM runtime's timestamping approach against a more traditional C program, we wrote a reactive pulse generator both in C and in SSLang; both implementations are shown in Figure 5.8. Each program attempts to match a 100 µs input pulse by "immediately" setting the output high upon seeing the rising edge, then setting the output low after 100 µs.

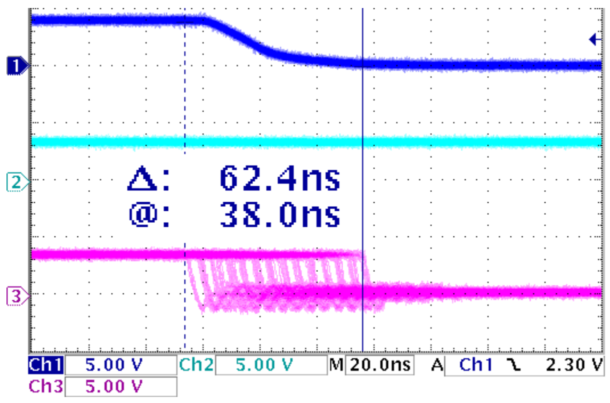
The C implementation (Figure 5.8a) performs all computation in the input ISR, including a 100 µs busy sleep. On the one hand, the C implementation represents the "best possible" pure software implementation, with no cycles spared on programming abstractions. On the other hand, the busy sleeping ISR prevents the processor from performing any other work between the rising and falling edge of the output pulse, meaning this approach cannot be used with concurrent real-time tasks on the same core.

Meanwhile, the SSLang implementation (Figure 5.8b) uses an instantaneous assign to trigger the rising edge of the output pulse, and schedules the falling edge **after** 100 µs via SSLang's deferred assignment mechanism. Since pulse is written in SSLang, this program can be run alongside other concurrent tasks, with the scheduling and external I/O events handled by SSLang's SSM runtime.

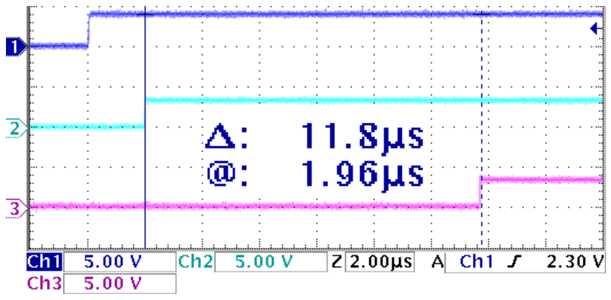
This experiment compares two important properties of our reactive program. First, the time it takes to produce the rising edge of the output pulse, using assign, represents its *response time*,



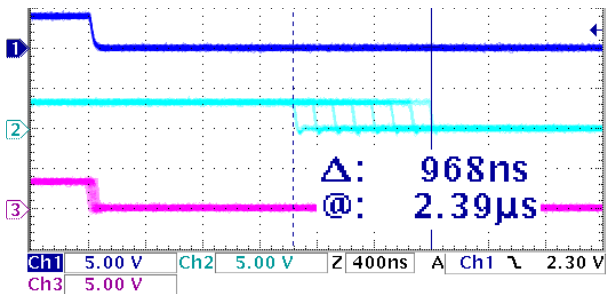
(a) The C and SSLang programs try to match the 100 µs input pulse.



(b) SSLang's falling edge is at most 62.4 ns late, and has less than 38.0 ns jitter.



(c) For the rising edge, C responds faster (1.96 µs) than SSLang (13.8 µs).



(d) C's falling edge is between 1.43 µs–2.39 µs late, and has around 2.39 µs jitter.

Figure 5.9: Behavior of the reactive pulse generators from Figure 5.8. Figure (a) shows the overall behavior, while Figures (b), (c), and (d) zoom into the rising and falling edges of the pulse. The top trace (blue) is the input; the middle (cyan) is the C program's output; the bottom (magenta) is from SSLang.

which is the latency incurred by the system as it emits an output in reaction to some input. Second, the time between the input and output pulses' falling edges is the *accuracy* of the system, how closely it adheres to the precise timing specified by **after** when given ample time to react.

Our measurements (Figure 5.9) show that the C program has a faster response time at first, but that the SSLang program times its falling edge far more accurately. Figure 5.9c, which is zoomed in on the rising edge of the pulses, shows that the C program has a shorter reaction time of 1.96 μs, compared to SSLang's 13.8 μs. Here, the C program's better performance is due to the fact that we eliminated most software overhead by performing all work in an input-triggered ISR; almost all of the 13.8 μs latency is due to the interrupt-handling latency of the RP2040 hardware.

Meanwhile, the SSLang program times the falling edge significantly better because of our PIO timestamp peripherals. The output of the C program is 1.43 μs–2.39 μs late (Figure 5.9d) because of the initial latency and imprecision in the busy wait loop, which polls the system timer. The SSLang program's falling output is 0 ns–62.4 ns late; that jitter, only visible in Figure 5.9b, is purely due to phase differences between the PIO's 16 MHz sampling clock and the frequency generator's oscillator. SSLang can achieve this level of precision—despite the overhead of its language runtime—because it reacts to events in terms of logical timestamps that are isolated from the system's internal physical execution time.

Chapter 6: Conclusion

The thesis of this dissertation is:

> *Temporal abstractions can be built using programming languages; programming languages can harness their platform's timing capabilities through temporal abstractions.*

To romanticize a little, temporal abstractions and programming languages bring out the best in each other, and empower developers to build portable and extensible real-time software.

This thesis is supported by my work on SSM, a programming model that brings temporal abstractions to untimed languages. My dissertation gave a formal account of this programming model by way of SSMΛ, and described its implementation in three languages, SSLang, Scoria, and ssm-lua. I explained how SSLang's runtime encapsulates its use of hardware timers and timestamp peripherals, and evaluated several small reactive applications that overcome SSM's abstraction penalty and achieve precise sub-microsecond timing on the RP2040 platform.

This chapter concludes my dissertation by highlighting some implications of this work (Section 6.1) and pointing out some avenues for future work (Section 6.2).

6.1 Implications

Timestamps and priorities are complementary mechanisms for execution order.

Simultaneity is an important aspect of real-time programming models. Timestamps can tell us whether one event occurs before another, but a system governed by logical time must find other ways to establish order when timestamps overlap. To avoid the pitfalls of asynchrony, SSM distinguishes between two levels of execution order: the order of instants, sorted by time, and process order within an instant, sorted by priority. This distinction arises in its step relation as two rules, S-Tick and S-Reduce (Figure 2.7), and manifests in our implementation as two priority queues, the event queue and the run queue (Figure 4.2). Using a separate mechanism for dealing with simultaneous events gives SSM its determinism.

Synchrony is compatible with conventional, untimed semantics.

Conventional models of computation are designed to be oblivious to the passage of wall-clock time. We can assimilate these foundations in a synchronous setting by treating untimed computation as the basis for execution within an instant, during which no logical time passes. The role of a timed programming model such as SSM is to *extend* this foundation, with mechanisms for terminating one instant (**wait**) and triggering the next (**after**) as well as an interpretation for simultaneous events. I demonstrated this point in Section 2.2 by presenting SSM's formal semantics through SSMΛ: the subset of SSMΛ without **wait**, **after**, **since**, and **par** is essentially a standard call-by-value lambda calculus, and many of SSMΛ's reduction rules (Figure 2.8) have direct analogues in ML-the-calculus [84].

Embedding is a pragmatic implementation strategy for synchronous models.

Synchronous programming models provide a formal basis for building temporal abstractions, but they must be actualized in a language that contributes other aspects of programmability. Developing such a language from scratch entails a non-trivial engineering burden. Although my work on SSM eventually culminated in SSLang, a standalone language, my dissertation also presented two embeddings of SSM in existing, general-purpose languages. First, I presented Scoria (Section 3.2), a deep embedding which helped us test what later became SSLang's language runtime. Then, I discussed the design and implementation of ssm-lua (Section 3.3), a shallow embedding used to prototype alternative SSM primitives. Together, these embeddings show it is feasible and pragmatic to directly implement synchrony within existing languages.

Embedded systems should delegate timestamping to hardware.

Software is better at building application logic than hardware, but its expressiveness comes at the cost of physical timing precision. Embedded systems can overcome this cost by delegating time-sensitive tasks to hardware, but they need abstractions that can precisely convey timing information between software and hardware. For instance, SSM uses scheduled variables to encapsulate peripherals that timestamp input events and scheduled timestamped output events. The system described in this dissertation implements timestamp peripherals using the RP2040's PIO system (Section 4.3) and achieves 62.5 ns precision, though similar results could be achieved with peripherals implemented in an FPGA or directly on the processor chip.

Determinism is useful for testing.

SSM has insisted on determinism from its inception, and uses unique process priorities to determine the order of execution within an instant. We found determinism to be invaluable during testing because it meant we only had to consider one valid behavior.[1] In particular, I have described two implications of determinism for testing. First, it meant we could compare our implementation against a simple reference interpreter. We did not need it to enumerate multiple possible behaviors, so we could be reasonably confident of its completeness from the outset. Second, it allowed us to synchronize and validate our runtime's platform driver with a simple correctness condition which we specified within the model.

[1] Technically, *at most* one behavior rather that *exactly* one because some programs should not have any valid behavior, e.g., those that contain type errors.

6.2 Future Work

SSM needs mechanisms for managing drift and handling errors.

We designed SSM to be an expressive and predictable real-time programming model, but it still lacks some useful features found in other models. In particular, SSM cannot detect and handle missed deadlines; Lingua Franca calls these "fault conditions" [70]. Following Janin's timed monad [59], we can add a **drift** primitive to SSM that tells a program how far it is running behind physical time. Doing so would introduce nondeterminism to our programming model, but that may be a necessary and reasonable compromise.

SSM also lacks features for process control and exception handling akin to Céu's "orthogonal abortion" mechanism [89]. Erlang's signals and monitors [19] may be a source of an inspiration.

SSM programs may benefit from a timing-aware type system.

SSM provides a framework for building temporal abstractions, but it does not impose a timing-aware type system—SSLang uses a conventional **let**-polymorphic type system that is not specialized for its programming model. I believe a well-designed temporal type system can complement sparse synchronous programming in two ways.

First, types can describe *how* processes should interact with scheduled variables rather than only *what* they may assigned. Basic examples include a read-only variable that processes may **wait** on but not assign to, or a variable that only accepts delayed assignments scheduled at least 5 ms in advance. A system incorporating stream types [24] or session types [48] may express more complex interactions with scheduled variables over time.

Second, types can encapsulate the behavior of temporal abstractions themselves. For instance, Jeffrey [60] has shown that type operators derived from linear temporal logic can be used to convey stateless and causal functions in FRP programs. Meanwhile, effect systems [67, 82] can distinguish functions that terminate instantly and functions that may suspend. Both kinds of type systems are promising bases for SSM.

SSM systems may defer work to idle periods.

SSM conveys a very precise model of when a system is active or idle. In fact, idle periods are abundant for the sparse workloads that SSM is designed for. The SSM runtime described in Chapter 4 simply sleeps during idle periods but a smarter implementation may use them for several kinds of deferred work. A natural choice is to perform garbage collection while the system is idle. Our SSM runtime currently manages heap-allocated objects with a naive reference counting scheme which adds overhead to each instant and cannot handle cyclic data structures. We can eliminate these issues with a deferred reference counting scheme [4, 5, 28] that processes deferred counts and collects cycles during SSM's idle periods.

Another idea is to defer non-urgent computations and evaluate them between instants. For example, when a program executes a delayed assignment **after** `d, x` $\leftarrow$ `e`, it does not need the value of `e` until later, so it can defer the evaluation of `e` as long as `e` has no side effects. We can take this idea further and embed SSM in a pure, non-strict language where any computation (that is not needed for output) can be deferred; a non-strict embedding may also benefit from infinite data structures and improved code modularity [50, 93].

SSM's scheduler might not need to advance logical time linearly.

Our SSM runtime's tick loop only advances logical time once the corresponding physical time has passed in case of then-unknown real-time input events that may interrupt the behavior of the system; this conservative approach means that each instant is always completed slightly late. Zou et al. minimize this lag by running parts of their system out of order and ahead of system time [101]. This kind of "time warping" execution strategy is implemented in PtidyOS [100] and its successor Lingua Franca [70], and exploits information about the minimum logical delay for input events to propagate to each downstream actor. The main challenge in applying this technique to SSM is that our model is imperative, so the topology of its dataflow graph is implicit and dynamic.

References

[1] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott, “A foundation for actor computation,” *Journal of Functional Programming*, vol. 7, no. 1, Jan. 1997.

[2] Andrew W. Appel and Zhong Shao, “An empirical and analytic study of stack vs. heap cost for languages with closures,” *Journal of Functional Programming*, vol. 6, no. 1, pp. 47–74, Nov. 1996.

[3] Lennart Augustsson, “Compiling pattern matching,” in *Proceedings of the 1958 Conference on Functional Programming Languages and Computer Architecture*, ser. LNCS 523, Nancy, France: Springer-Verlag, Sep. 1985, pp. 368–381.

[4] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith, “Java without the coffee breaks: A nonintrusive multiprocessor garbage collector,” in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, USA: Association for Computing Machinery, 2001, pp. 92–103.

[5] David F. Bacon and V. T. Rajan, “Concurrent cycle collection in reference counted systems,” in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. LNCS 2072, Budapest, Hungary: Springer, Jun. 2001, pp. 207–235.

[6] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito, “Two simplified algorithms for maintaining order in a list,” in *Proceedings of the 10th Annual European Symposium on Algorithms*, Rolf Möhring and Rajeev Raman, Eds., ser. LNCS 2461, Rome, Italy: Springer-Verlag, Sep. 2002, pp. 152–164.

[7] Jon Bentley, “Programming pearls: Associative arrays,” *Communications of the ACM*, vol. 28, no. 6, pp. 570–576, Jun. 1985.

[8] Albert Benveniste and Gérard Berry, “The synchronous approach to reactive and real-time systems,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep. 1991.

[9] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone, “The synchronous languages 12 years later,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[10] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot, “The SIGNAL software environment for real-time system specification, design, and implementation,” in *IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, 1989, pp. 41–49.

[11] Gérard Berry and Georges Gonthier, “The ESTEREL synchronous programming language: Design, semantics, implementation,” *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[12] Andrew P. Black, Magnus Carlsson, Mark P. Jones, Dick Kieburtz, and Johan Nordlander, “Timber: A programming language for real-time embedded systems,” Oregon Health & Science University, Tech. Rep. CSE-02-002, Apr. 2002.

[13] Frédéric Boussinot, “Reactive C: An extension of C to program reactive systems,” *Software: Practice and Experience*, vol. 21, no. 4, pp. 401–428, Apr. 1991.

[14] Frédéric Boussinot and Robert de Simone, “The ESTEREL language,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, Sep. 1991.

[15] Frédéric Boussinot and Robert de Simone, “The SL synchronous language,” *IEEE Transactions on Software Engineering*, vol. 22, no. 4, pp. 256–266, Apr. 1996.

[16] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella, “HOPE: An experimental applicative language,” in *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, Stanford, CA, USA, Aug. 1980, pp. 136–143.

[17] Magnus Carlsson, Johan Nordlander, and Dick Kieburtz, “The semantic layers of Timber,” in *Proceedings of the 1st Asian Symposium on Programming Languages and Systems*, ser. LNCS 2895, Beijing, China: Springer-Verlag, Nov. 2003, pp. 339–356.

[18] Manuel Carro, José F. Morales, Henk L. Muller, Germán Puelba, and Manuel V. Hermenegildo, “High-level languages for small devices: A case study,” in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Seoul, Korea: Association for Computing Machinery, Oct. 2006, pp. 271–281.

[19] Aurélie Kong Win Chang, Jérôme Feret, and Gregor Gössler, “A semantics of core Erlang with handling of signals,” in *Proceedings of the 22nd ACM SIGPLAN International Workshop on Erlang*, ser. Erlang 2023, Seattle, WA, USA: Association for Computing Machinery, Sep. 2023, pp. 31–38.

[20] Alonzo Church and J. Barkley Rosser, “Some properties of conversion,” *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482, May 1936.

[21] Koen Claessen and John Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada: Association for Computing Machinery, Sep. 2000, pp. 268–279.

[22] George E. Collins, “A method for overlapping and erasure of lists,” *Communications of the ACM*, vol. 3, no. 12, pp. 655–657, Dec. 1960.

[23] Antony Courtney, Henrik Nilsson, and John Peterson, "The Yampa arcade," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '03, Uppsala, Sweden: Association for Computing Machinery, Aug. 2003, pp. 7–18.

[24] Joseph W. Cutler *et al.*, "Stream types," *Proceedings of the ACM on Programming Languages*, PACMPL, vol. 8, no. PLDI, Art. no. 204, Jun. 2024.

[25] Evan Czaplicki and Stephen Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, USA: Association for Computing Machinery, Jun. 2013, pp. 411–422.

[26] Luís Damas and Robin Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, NM, USA: Association for Computing Machinery, Jan. 1982, pp. 207–212.

[27] Ana Lúcia de Moura and Roberto Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 2, Art. no. 6, Feb. 2009.

[28] L. Peter Deutsch and Daniel G. Bobrow, "An efficient, incremental, automatic garbage collector," *Communications of the ACM*, vol. 19, no. 9, pp. 522–526, Sep. 1976.

[29] Paul F. Dietz and Daniel D. Sleator, "Two algorithms for maintaining order in a list," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, New York, NY, USA, Jan. 1987, pp. 365–372.

[30] Stephen A. Edwards, "On determinism," in *Principles of Modeling: Essays dedicated to Edward A. Lee on the Occasion of his 60th Birthday*, ser. LNCS 10760, Patricia Derler, Marten Lohstroh, and Marjan Sirjani, Eds., Berkeley, CA, USA: Springer, Oct. 2017, pp. 240–253.

[31] Stephen A. Edwards and John Hui, "The sparse synchronous model," in *Proceedings of the 2020 Forum on Specification and Design Languages*, Kiel, Germany, Sep. 2020, Art. no. 2.1.

[32] Eric Eide and John Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM International Conference on Embedded Software*, Atlanta, GA, USA: Association for Computing Machinery, Oct. 2008, pp. 255–264.

[33] Johan Eker *et al.*, "Taming heterogeneity – the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[34] Conal Elliott, "Compiling to categories," *Proceedings of the ACM on Programming Languages*, PACMPL, vol. 1, no. ICFP, Art. no. 27, Sep. 2017.

[35] Conal Elliott and Paul Hudak, "Functional reactive animation," in *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, The Netherlands: Association for Computing Machinery, Jun. 1997, pp. 263–273.

[36] Conal M. Elliott, "Push-pull functional reactive programming," in *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Edinburgh, Scotland: Association for Computing Machinery, Sep. 2009, pp. 25–36.

[37] Matthias Felleisen, "On the expressive power of programming languages," in *Proceedings of the 3rd European Symposium on Programming*, ser. LNCS 432, Copenhagen, Denmark: Springer-Verlag, May 1990, pp. 134–151.

[38] Matthias Felleisen and Daniel P. Friedman, "Control operators, the SECD-machine, and the λ-calculus," in *Proceedings of the 3rd IFIP Working Conference on Formal Description of Programming Concepts*, Ebberup, Denmark: North-Holland, Aug. 1986, pp. 193–222.

[39] Matthias Felleisen and Daniel P. Friedman, "A calculus for assignments in higher-order languages," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Munich, West Germany: Association for Computing Machinery, Jan. 1987, pp. 314–325.

[40] OpenJS Foundation, *The Node.js event loop, timers, and process.nextTick()*, [Online]. Available: `https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick`, 2016.

[41] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[42] Nicolas Halbwachs, "A synchronous language at work: The story of Lustre," in *Formal Methods for Industrial Critical Systems*. John Wiley & Sons, Ltd, 2012, ch. 2, pp. 15–31.

[43] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler, "Type classes in haskell," *ACM Transactions on Programming Languages and Systems*, TOPLAS, vol. 18, no. 2, pp. 109–138, 1996.

[44] David Harel and Amir Pnueli, "On the development of reactive systems," in *Proceedings of the NATO Advanced Study Institute on Logics and Models of Concurrent Systems*, ser. NATO ASI F, vol. 13, La Colle-sur-Loup, France: Springer-Verlag, Oct. 1984, pp. 477–498.

[45] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

[46] Carl Hewitt, Peter Bishop, and Richard Steiger, “A universal modular ACTOR formalism for artificial intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 1973, pp. 235–245.

[47] Charles A. R. Hoare, “Monitors: An operating system structuring concept,” *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.

[48] Kohei Honda, Nobuko Yoshida, and Marco Carbone, “Multiparty asynchronous session types,” *Journal of the ACM*, vol. 63, no. 1, Art. no. 9, Mar. 2016.

[49] Paul Hudak, “Building domain-specific embedded languages,” *ACM Computing Surveys*, vol. 28, no. 4es, Art. no. 196, Dec. 1996.

[50] John Hughes, “Why functional programming matters,” *The Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[51] John Hui, Kyle J. Edwards, and Stephen A. Edwards, “Timestamp peripherals for precise real-time programming,” in *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for Codesign*, Hamburg, Germany: Association for Computing Machinery, Oct. 2023, pp. 137–147.

[52] John Hui and Stephen A. Edwards, “Towards sparse synchronous programming in Lua,” in *Proceedings of Cyber-Physical Systems and Internet of Things Week (CPS-IoT Week), 1st Workshop on Time-Centric Reactive Software*, San Antonio, TX, USA: Association for Computing Machinery, May 2023, pp. 361–366.

[53] John Hui and Stephen A. Edwards, “The sparse synchronous model on real hardware,” *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 5, Art. no. 69, Sep. 2024.

[54] *IEEE standard for floating-point arithmetic*, IEEE Standard 754-2019, IEEE Computer Society, New York, NY, USA, 2019.

[55] *IEEE standard hardware description language based on the Verilog hardware description language*, IEEE Standard 1364-1995, IEEE Computer Society, New York, NY, USA, 1996.

[56] *IEEE standard VHDL reference manual (1076–1987)*, IEEE Standard (1076–1987), IEEE Computer Society, New York, NY, USA, 1988.

[57] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, “The evolution of Lua,” in *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, San Diego, California, Jun. 2007, Art. no. 2.

[58] Ivan Manuel Bärenz Ivan Perez and Henrik Nilsson, “Functional reactive programming, refactored,” in *Proceedings of the 9th International Symposium on Haskell*, Nara, Japan: Association for Computing Machinery, Sep. 2016, pp. 33–44.

[59] David Janin, “A timed IO monad,” in *Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages*, ser. LNCS 12007, New Orleans, LA, USA: Springer-Verlag, Jan. 2020, pp. 131–147.

[60] Alan Jeffrey, “LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs,” in *Proceedings of the 6th Workshop on Programming Languages Meets Program Verification*, Philadelphia, PA, USA: Association for Computing Machinery, Jan. 2012, pp. 49–60.

[61] Philip J. Kiviat, “Development of discrete digital simulation languages,” *SIMULATION*, vol. 8, no. 2, pp. 65–70, 1967.

[62] Neelakantan R. Krishnaswami, “Higher-order functional reactive programming without spacetime leaks,” in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, Boston, MA, USA: Association for Computing Machinery, Sep. 2013, pp. 221–232.

[63] Robert Krook, John Hui, Bo Joel Svensson, Stephen A. Edwards, and Koen Claessen, “Creating a language for writing real-time applications for the internet of things,” in *Proceedings of the 20th ACM-IEEE International Conference on Formal Methods and Models for Codesign*, Shanghai, China: Association for Computing Machinery, Oct. 2022, pp. 78–97.

[64] Edward A. Lee, “Modeling concurrent real-time processes using discrete events,” *Annals of Software Engineering*, vol. 7, pp. 25–45, Oct. 1999.

[65] Edward A. Lee, “Computing needs time,” *Communications of the ACM*, vol. 52, no. 5, pp. 70–79, May 2009.

[66] Edward A. Lee, “Determinism,” *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, Art. no. 38, May 2021.

[67] Daan Leijen, “Type directed compilation of row-typed algebraic effects,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris, France: Association for Computing Machinery, Jan. 2017, pp. 486–499.

[68] Xavier Leroy, *Compiling functional languages*, Presentation at Spring School “Semantics of Programming Languages”, Agay, France, Mar. 2002.

[69] Marten Lohstroh and Edward A. Lee, “Deterministic actors,” in *Proceedings of the 2019 Forum on Specification and Design Languages*, Southampton, United Kingdom, Sep. 2019, Art. no. 2.1.

[70] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee, “Toward a lingua franca for deterministic concurrent systems,” *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 4, Art. no. 36, Jul. 2021.

[71] Louis Mandel, Cédric Pasteur, and Marc Pouzet, “ReactiveML, ten years later,” in *Proceedings of the 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Siena, Italy: Association for Computing Machinery, Jul. 2015, pp. 6–17.

[72] Louis Mandel and Marc Pouzet, “ReactiveML, a reactive extension to ML,” in *Proceedings of the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Lisbon, Portugal: Association for Computing Machinery, Jul. 2005, pp. 82–93.

[73] Nikhil Marathe. “libuv design overview.” (2012).

[74] Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu, “Embedding by unembedding,” *Proceedings of the ACM on Programming Languages*, PACMPL, vol. 7, no. ICFP, Art. no. 189, Aug. 2023.

[75] Alex McLean, “Making programming languages to dance to: Live coding with Tidal,” in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, Gothenburg, Sweden: Association for Computing Machinery, Sep. 2014, pp. 63–70.

[76] Henrik Nilsson, Antony Courtney, and John Peterson, “Functional reactive programming, continued,” in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Pittsburgh, PA, USA: Association for Computing Machinery, Oct. 2002, pp. 51–64.

[77] Johan Nordlander, Mark P. Jones, Magnus Carlsson, Richard B. Kieburtz, and Andrew P. Black, “Reactive objects,” *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pp. 155–158, 2002.

[78] *NRF52840 product specification*, v1.7, Nordic Semiconductor ASA, Nov. 2021.

[79] Thierry Gautier Paul Le Guernic, Michel Le Borgne, and Claude Le Maire, “Programming real-time applications with SIGNAL,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, Sep. 1991.

[80] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller, “Copilot: A hard real-time runtime monitor,” in *Proceedings of the 1st International Conference on Runtime Verification*, ser. LNCS 6418, St. Julians, Malta: Springer, Nov. 2010, pp. 345–359.

[81] Gordon D. Plotkin, “Call-by-name, call-by-value and the λ-calculus,” *Theoretical Computer Science*, vol. 1, no. 2, pp. 125–159, Dec. 1975.

[82] Gordon D. Plotkin and Matija Pretnar, “Handling algebraic effects,” *Logical Methods in Computer Science*, vol. 9, no. 4, Art. no. 23, Dec. 2013.

[83] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry, *Compiling Esterel.* Springer, May 2007.

[84] François Pottier and Didier Rémy, “The essence of ML type inference,” in *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce, Ed., Cambridge, MA, USA: MIT Press, 2004, ch. 10.1, pp. 387–406.

[85] libuv project, *libuv*, Available: `https://github.com/libuv/libuv`, 2012.

[86] Zephyr Project, *Zephyr RTOS*, Available: `https://github.com/zephyrproject-rtos/zephyr`, 2016.

[87] *RP2040 datasheet*, Raspberry Pi Ltd, Cambridge, United Kingdom, 2023.

[88] Francisco Sant’anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto, and Adriano Branco, “The design and implementation of the synchronous language Céu,” *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 4, Art. no. 98, Nov. 2017.

[89] Rodrigo C. M. Santos, Guilherme F. Lima, Francisco Sant’Anna, Roberto Ierusalimschy, and Edward H. Haeusler, “A memory-bounded, deterministic and terminating semantics for the synchronous programming language Céu,” in *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2018, pp. 1–18.

[90] Julian Seward and Nicholas Nethercote, “Using Valgrind to detect undefined value errors with bit-precision,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Anaheim, CA, USA: USENIX Association, Apr. 2005, pp. 17–30.

[91] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden, “Practical causality handling for synchronous languages,” in *Proceedings of Design, Automation, and Test in Europe*, Florence, Italy, Mar. 2019, pp. 1281–1284.

[92] Wouter Swierstra, “Data types à la carte,” *Journal of Functional Programming*, vol. 18, no. 4, pp. 423–436, Mar. 2008.

[93] David A. Turner, “The semantic elegance of applicative languages,” in *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, NH, USA: Association for Computing Machinery, Oct. 1981, pp. 85–92.

[94] Reinhard von Hanxleden *et al.*, "Sequentially constructive concurrency—a conservative extension of the synchronous model of computation," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4s, Art. no. 144, Jul. 2014.

[95] Willim W. Wadge and Edward A. Ashcroft, *LUCID, the dataflow programming language*. USA: Academic Press Professional, Inc., 1985.

[96] Philip Wadler, "Efficient compilation of pattern-matching," in *The Implementation of Functional Programming Languages* (Prentice-Hall International series in computer science), Simon Peyton Jones, Ed., Prentice-Hall International series in computer science. Hemel Hempstead, United Kingdom: Prentice Hall International, 1987, ch. 5, pp. 78–103.

[97] Philip Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, Nice, France: Association for Computing Machinery, 1990, pp. 61–78.

[98] Reinhard Wilhelm *et al.*, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, May 2008.

[99] Yang Zhao, Jie Liu, and Edward A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, Bellevue, WA, United States, Apr. 2007, pp. 259–268.

[100] Jia Zou, Slobodan Matic, and Edward A. Lee, "PtidyOS: A lightweight microkernel for Ptides real-time systems," in *Proceedings of the 18th IEEE Real Time and Embedded Technology and Applications Symposium*, Beijing, China, Apr. 2012, pp. 209–218.

[101] Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng, and Patricia Derler, "Execution strategies for PTIDES, a programming model for distributed embedded systems," in *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, USA, Apr. 2009, pp. 77–86.