The Sparse Synchronous Model on Real Hardware

JOHN HUI and STEPHEN A. EDWARDS, Columbia University, USA

We present the Sparse Synchronous model (SSM) of computation, which allows a programmer to specify software timing more precisely than the traditional "heartbeat" of mainstream operating systems or the synchronous languages. SSM is a mix of semantics inspired by discrete event simulators and the synchronous languages designed to operate in resource-constrained environments such as microcontrollers. SSM provides precise timing prescriptions, concurrency, and determinism. We implement SSM in SSML, a toy language along with a runtime system that includes a scheduler, memory manager, and an interface that works with an RTOS to keep the model synchronized with the real world. Experimentally, we find our implementation is able to perform jitter-free I/O in the 10s of kHz on a microcontroller.

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

Additional Key Words and Phrases: real time systems, concurrency control, computer languages, timing

1 INTRODUCTION

A colleague, who trains rats to perform simple tasks, needed control over stimuli timing and measurement of response timing. The usual ad hoc solution of writing C programs for a microcontroller with timers requires a sophisticated programmer (e.g., not the typical biologist) and is difficult to maintain across different hardware. Our colleague had moved to a microcontroller running a cyclic executive that simulated a finite state machine stored in an array, but found this model limiting and the timing precision insufficient. Meanwhile, rats are not periodic enough for a real-time operating system (RTOS) or the sample-driven implementation style typical of the synchronous languages [4].

Our Sparse Synchronous Model (SSM), an earlier version of which we presented elsewhere [12], was designed to address these needs. Our goals were precise (μs-level) timing specification and measurement, deterministic concurrency, and platform-speed-independent I/O. We call the model "sparse" because its synchronous execution model is not driven by a periodic timer, and supports advancing time by arbitrary increments between instants of computation. Since then, we have added functional language features such as recursion, algebraic data types, and automatic memory management, and validated our technique on real hardware under existing RTOSes.

Consider the signal generator program shown in Figure 1, which is written in SSML, the new language we present here. This program generates a square wave signal whose frequency can be adjusted by two buttons. The entry point of the program, *main* (lines 12–15), takes handles for the two buttons (*button1* and *button2*) and a GPIO pin (*out*). The *main* function creates a shared variable *hPeriod* (line 13) that determines the half-period of the square wave, and runs *sigGen* and *sigCtl* concurrently (lines 14–15).

The *sigGen* function (lines 1–5) generates a precisely timed square wave on the GPIO pin by scheduling future output events and blocking on them. It consists of an infinite loop that schedules a toggle update to the GPIO pin (lines 3–4), then blocks until the update occurs (line 5). In line 4, the **after** directive schedules the update

Authors' address: John Hui, j-hui@cs.columbia.edu; Stephen A. Edwards, sedwards@cs.columbia.edu, Columbia University, New York, New York, USA, 10027.

1539-9087/2022/12-ART $15.00
https://doi.org/10.1145/3572920

```
 1 sigGen out hPeriod =
 2   while True do
 3     let delay = deref hPeriod ;                     // How long to sleep
 4     after delay, out ← not (deref out) ;            // Toggle value of out
 5     wait out                                        // Block until out is updated

 6 sigCtl button1 button2 hPeriod =
 7   while True do
 8     wait button1 | button2 ;                        // Wait for either button to be pressed
 9     if written button1 == now                       // Was button 1 pressed?
10       then hPeriod ← deref hPeriod ∗ 2              // Double the half-period
11       else hPeriod ← max (deref hPeriod / 2) 1      // Halve the half-period

12 main button1 button2 out =
13   let hPeriod = ref (usec 1000);                    // Shared variable
14   par sigGen out hPeriod                            // Generate a square wave
15     & sigCtl button1 button2 hPeriod                // and watch buttons
```

Fig. 1. A signal generator program. *sigGen* generates a square wave on the *out* signal whose period is set by *hPeriod*, which can be doubled and halved by pressing the *button1* or *button2* inputs.

according to the time of each loop iteration, independent of processor speed. Meanwhile, *sigCtl* waits for input from either of the two buttons (line 8) and doubles or halves the *hPeriod* variable depending on which button was pressed (lines 9–11).

Splitting up *sigGen* and *sigCtl* is a natural way to divide the two responsibilities of the program and improves code modularity. They run concurrently; neither terminates. SSM's deterministic semantics guarantee the absence of data races, in contrast to non-synchronous languages that typically require explicit yet error-prone synchronization to ensure determinism.

Our primary goal was the precise specification and measurement of real-time events. Our model treats time as a first-class object, like a discrete-event simulator. Inspired by the synchronous languages [4] and Ptides [31], an SSM system operates according to model time, which advances in discrete instants. In lines 2–5, the delay between *sigGen*'s toggle events is exactly the time set by the *hPeriod* variable—model time does not advance outside of blocking statements like **wait**.

Model time is discrete and not dense, precluding Zeno-like behavior because there is ultimately a smallest time step. For platform-independence, the fundamental time quantum is not visible to SSM programs. An SSM system only manipulates physical time units (seconds, milliseconds, etc.): the runtime is responsible for converting the microseconds (**usec**) specified in line 13 to the corresponding number of ticks used by each platform's timing hardware.

Our runtime system isolates an SSM program from the passage of physical (wall-clock) time, preserving the meaning of SSM programs across platforms. For both timing precision and efficiency, SSM does not use the periodic sampling approach typically adopted by synchronous language implementations, where the system performs computation in steps triggered by, say, a 10 ms periodic timer. Though this sample-driven approach is straightforward to implement, the period must be long enough to accommodate work performed in each instant, limiting timing precision. Instead, our interrupt-driven runtime sleeps until the next active instant, using a precise hardware timer to synchronize with physical time. With our approach, the frequency of our signal generator is not limited by a low-speed periodic timer, and also avoids unnecessary wake-ups when a lower frequency is requested.

SSM provides deterministic concurrency by totally ordering the execution of concurrent tasks within an instant. Specifically, it uses cooperative multitasking with a programmer-prescribed order: in the signal generator program, *sigGen* takes priority over *sigCtl* when they run in the same instant, because *sigGen* appears first in the **par** invocation in lines 14–15. By contrast, discrete-event models are usually nondeterministic, as they erroneously treat simultaneous events (with identical timestamps) as order-independent. While simply prohibiting simultaneous events might seem an attractive solution, simultaneity seems to be inherent to concurrent systems.

We wanted recursive function calls, so SSM is built around function activation records that are created and destroyed as SSM routines are called and return. In addition to local variables and linkage to its caller, a routine's activation record stores its control state when it is suspended waiting for an event and bookkeeping used by the runtime system to determine when to resume.

In this paper, we present SSML, a toy language that embodies SSM semantics (Section 2). We describe our compilation scheme (Section 3) and runtime implementation (Section 4), and discuss how we embedded this runtime in an existing real-time operating system to interact with the environment (Section 5). We evaluate the performance of our system on real hardware (Section 6) and offer comparisons to related work (Section 7).

The source code for our runtime system is available at https://github.com/ssm-lang/ssm-runtime.

2 SEMANTICS

2.1 Informal presentation

We illustrate the semantics of the Sparse Synchronous Model through "SSML," a toy language with functional features such as immutable data and algebraic data types, but no first-class functions or closures; imperative features such as mutable references, assignments, and loops; and synchronous features such as blocking parallel evaluation and delayed assignment. Functions defined in SSML are not necessarily pure; they are allowed to incur side effects, such as assigning to references or allowing model time to advance. Figure 2 is a contrived recursive Fibonacci example in SSML; Figure 3 shows the abstract syntax of the language; Figure 4 lists some built-in types and functions.

The Fibonacci example in Figure 2 illustrates many of the novel features of SSML: mutable references, scheduled future updates to references, blocking waits on such updates, and parallel evaluation. Control starts at *main* in line 12, which begins by using the built-in **ref** function to create a new mutable integer value referred to as *r* with an initial value of 0. These mutable references function as synchronized communication channels among concurrent threads. The *main* function then invokes the *fib* function in line 13, with arguments that represent the Fibonacci number to be computed and the reference *r*, where the result will be placed. Normal values, such as *n*, are immutable and passed by value; mutable references can both be updated immediately and scheduled to be updated later. Once *fib* terminates, *main* waits for *r* to be written (*fib* schedules a future update to *r* before it terminates), then calls *print* to display the value held by reference *r*, the **deref** function returns the current value of a reference.

The *fib* function in lines 4–10 takes two arguments *n* and *r*. In the base case, where *n* is less than 2, *fib* schedules a 1 to be written to *r* exactly one second after the *fib* function was invoked. Scheduling a future update to a reference is one of SSML's key temporal primitives. The *fib* function then terminates instantly. SSML adopts the synchronous model of time in which everything except **wait** is treated as running in exactly zero time [4].

In the recursive case, *fib* creates two new references *r1* and *r2* (lines 8 and 9) before spawning the execution of three functions in parallel: two recursive calls to *fib*, which will place their results in *r1* and *r2*, and a call to the *sum* function to add and return their results. The **par** does not terminate until all three of its parallel branches

```
1  sum r1 r2 r =                    // Ref Int -> Ref Int -> Ref Int -> ()
2    par wait r1 & wait r2 ;        // wait for both r1 and r2
3    after sec 1, r ← deref r1 + deref r2

4  fib n r =                        // Int -> Ref Int -> ()
5    if n < 2 then
6      after sec 1, r ← 1           // Base case, assigned after 1 second
7    else
8      let r1 = ref 0 ;
9      let r2 = ref 0 ;
10     par fib (n−1) r1 & fib (n−2) r2 & sum r1 r2 r

11 main n =                         // Int -> ()
12   let r = ref 0 ;
13   fib n r ;
14   wait r ;
15   print (deref r)
```

Fig. 2. A contrived Fibonacci example in SSML with delayed assignment (**after**), waiting for variable writes (**wait**), and parallel evaluation (**par**).

have terminated. This ability to spawn multiple, concurrent threads of control is another key temporal feature of SSML.

The threads forked by a **par** are evaluated left-to-right in each instant, ensuring that if a reference is written in a thread, the change is immediately visible to all threads to its right. However, because the *fib* threads communicate with *sum* through a delayed assignment, this detail does not affect this example's result.

The *sum* function waits for both the *r1* and *r2* references to be written (line 2), then schedules their sum to be written to *r* after one second and terminates instantly. Since *r1* and *r2* are references, their values are obtained by applying the **deref** built-in function.

Figure 4 lists types and functions defined in the standard library. Programs consist of algebraic type and function definitions. Algebraic data types (ADTs) are polymorphic and follow the basic ADTs of ML-family languages, including OCaml and Haskell: a type constructor starts with an uppercase letter (e.g., *Bool*) with zero or more type variables (that start with a lowercase letter) as arguments. Each type constructor defines one or more data constructors (which also start with an uppercase letter, e.g., *True*), each associated with zero or more payload fields of some specified type. Types (e.g., of payload fields) can be a data constructor passed zero or more types as arguments, a function type written with an infix ⟶ , or a type variable. Types may be polymorphic, so

```
type List a = Cons a (List a) | Nil
```

is the usual polymorphic List type; List Bool is a list of Booleans, List Int is a list of integers, and List (List Int) is a list of integer lists.

Figure 3 divides SSML expressions (which always produce a value) into four groups. The first group describes a pure functional language with sequential evaluation: expressions are data "variables" (lowercase names, e.g., *foo*); data constructors (uppercase names, e.g., *True*), which must have all their arguments provided, if any; literals (e.g., *42*); binary operations (e.g., *a + 3*); function calls, which must have all their arguments provided (e.g., *foo 3 4*); local variable definitions, written with a semicolon *;* to emphasize that the new variable's value is evaluated

program	::=	[*type-def* | *id* [*id*]$^+$ = *expr*]*	
type-def	::=	**type** *tcon-id tvar-id** = *dcon-id type** [**|** *dcon-id type**]*	Algebraic data type
type	::=	*tcon-id* [*type*]* | *type* **⟶** *type* | *tvar-id*	
expr	::=	*id*	Data variable (lowercase)
	|	*dcon-id* [*expr*]*	Data constructor (uppercase)
	|	*literal*	Literal constant, e.g., 42
	|	*expr un-op expr*	Unary operators
	|	*expr bin-op expr*	Binary operators
	|	*id* [*expr*]$^+$	Function Call
	|	**let** *id* = *expr* ; *expr*	Let-definition
	|	**match** *expr* **with** *pattern* **⟶** *expr* [**|** *pattern* **⟶** *expr*]*	Pattern match
	|	**if** *expr* **then** *expr* **else** *expr*	Conditional
	|	**par** *id* [*expr*]$^+$ [**&** *id* [*expr*]$^+$]*	Parallel evaluation
	|	**ref** *expr*	Create a new reference with initial value
	|	**deref** *expr*	Return a reference's value
	|	**written** *expr*	Return the last model time a reference was written
	|	*expr* **⟵** *expr*	Instantaneous assignment to reference
	|	*expr* ; *expr*	Sequencing
	|	**while** *expr* **do** *expr*	Loop
	|	**after** *expr* **,** *expr* **⟵** *expr*	Delayed assignment
	|	**wait** *expr* [**|** *expr*]*	Suspend waiting on references
	|	**now**	The current model time
pattern	::=	*dcon-id* [*pattern*]* | *id* | _	

Fig. 3. Abstract syntax of SSML. Brackets [], bars |, asterisks *, pluses $^+$, and question marks $^?$ denote syntactic grouping, choice, zero-or-more, one-or-more, and zero-or-one. Tokens are bold.

type () = ()	The unit type and its one data constructor
type Bool = **True** | **False**	The Boolean type
type Int = **0** | **1** | **2** | ···	The integer type
type Time = . . .	The abstract model time type
type Ref a = . . .	The polymorphic reference type
sec *expr*	Treat number as a number of seconds : *Int* ⟶ *Time*
msec *expr*	Treat number as a number of milliseconds : *Int* ⟶ *Time*
usec *expr*	Treat number as a number of microseconds : *Int* ⟶ *Time*

Fig. 4. SSML built-in types and functions.

before the body (e.g., **let** *a* = *foo 3* ; *a* + *1* evaluates *foo 3*, binds it to *a*, then evaluates and produces *a* + *1*); and pattern matches: multiway conditional constructs that evaluate an expression, compare the data constructor of

```
1 timeout2 t a b =
2   let tt = ref () ;
3   after t, tt ← () ;  // Schedule a pure event
4   wait a | b | tt     // Wait for a, b, or tt

5 main env =
6   let a = new 0 ;
7   let b = new 0 ;
8   timeout2 (sec 3) a b
```

Fig. 5. Expressing timeout behavior. Since a **wait** statement resumes as soon as *any* of its variables are written, adding a "timeout" variable *tt* and scheduling it to be written in the future just before a **wait** effects a timeout mechanism. Here, since *a* and *b* are not written beyond initialization, *main* will resume after 3 seconds and both **written** *a* and **written** *b* will be 0, indicating they did not cause the timeout.

the resulting value against the patterns, bind the payload fields to the named field identifiers (or discard them in the case of _), and then evaluate the associated expression.

The **par** construct evaluates multiple function calls in parallel and terminates when all of them have been evaluated. The context where each function call is evaluated is called a *process*. Within each instant, the various branches of a **par** are executed in order from left to right, ensuring any side-effects (described below) are evaluated in a deterministic order.

Most SSML values are immutable, but like other ML-family languages, SSML has references to mutable values. References are obtained and initialized via allocation (**ref**), read via dereference (**deref**), and written to via assignment (←). References are used as communication channels between concurrent processes. Like condition variables in traditional threaded code, references in SSML "announce" when they are written, and wake up processes that suspended to wait on them (described below). However, unlike threaded code with condition variables, concurrent computation in SSML is totally ordered, ruling out data races.

SSML provides two primitives for temporal control: **after**, which schedules a delayed assignment to a reference; and **wait**, which suspends the calling process until at least one of a set of references is written to.

A delayed assignment (**after**) schedules a particular reference to be assigned to in a later instant; the delayed assignment itself does not take any model time to evaluate. In Figure 2, both **after** statements (in lines 3 and 6) schedule an update to *r* one second in the future. Time delays may not be zero (normal assignment statements are used for this) or negative. SSML only allows one outstanding update per reference; an update overwrites any pending update. This design avoids an unbounded accumulation of updates and eliminates nondeterminism that could arise from instants with multiple scheduled updates to the same reference.

A **wait** expression causes the current function to suspend execution in the current instant and reawaken in the next instant in which any of the given references have been written. Unlike discrete-event languages like VHDL [28] designed for digital logic simulation, SSML routines are awakened by *any* write to a reference, not just writes that change the reference's value. We chose the event-on-write policy because we wanted to make events explicit, rather than merely using them to model continuous behavior. Our policy enables us to model pure events through variables that only take a single value, "unit," written as (), and to allow variables to convey sequences of values without two identical values in sequence being inadvertently merged. For instance, in Figure 5, the *tt* reference conveys a pure event to signal a timeout for the wait in line 4. SSML can still express VHDL's event-on-change policy by enclosing each assignment in a conditional that only writes to a reference if its updated value differs from its previous one.

wait expressions are the only ones that directly advance model time; others may incur delays if they cause a **wait** to be invoked. For instance, a function call will block until the callee returns, so model time will pass if the callee waits, e.g., *timeout2* in Figure 5. All other expressions terminate in the instant they were started. SSML adopts the synchronous hypothesis, which insists all instants are evaluated atomically, so that SSML programs' specified real-time behavior do not depend on the timing characteristics of the platform they are executed on.

```
1  foo a =
2    wait a ;
3    a ← a * 2  // Runs at 1 second, but after bar

4  bar a =
5    wait a ;
6    a ← a + 4  // Runs at 1 second, before foo

7  main env =
8    let a = ref 0 ;
9    after sec 1, a ← 1 ;
10   par bar a & foo a    // bar will run before foo
11   print (deref a)      // 10
```

Fig. 6. Children of a *fork* always execute in order: at 1 second, a write to *a* awakens routines *first* and *second*, which execute in that order.

The **written** function returns the model time at which a reference was last written and **now** evaluates to the model time of the current instant. When a process unblocks from waiting on multiple references, testing **written** *r* = **now** indicates whether *r* was responsible for waking the process.

While **par** expressions are conjunctive—they block until *all* the called functions have terminated—**wait** expressions are disjunctive—they resume when *any* of their references are written. This distinction explains why we evaluate two **wait** expressions in parallel in Figure 2 (line 2): **wait** *r1* | *r2* would have terminated after only the first of *r1* and *r2* had arrived, whereas *sum* wants the new values of both *r1* and *r2*. Line 2 also behaves as desired when *r1* and *r2* arrive simultaneously.

In each instant, the children of a parallel expression are executed in the order they are listed. Figure 6 illustrates this policy: at 1 second, the delayed assignment to *a* will wake up both *foo* and *bar*. However, because *bar* appears before *foo* in the par expression on line 10, *bar* will run first, reading *a*'s new value of 1 and changing it to 5. Then, *foo* will run, multiplying *a* by 2 to produce 10.

To perform I/O, SSML programs interact with *input* and *output references*. These references are given to *main* as parameters and abstract the external environment, akin to how C programs may communicate with hardware peripherals via mapped memory. The runtime is responsible for conveying the environment via these references: input references may be externally updated at the beginning of an instant, while output references produce some external side effect (invisible to the SSML program) when assigned to. Otherwise, these references behave the same as regular SSML references.

2.2 Formal semantics

SSML is a call-by-value, pass-by-value functional language; we formally present the semantics of SSML as a term rewriting system in the style of Crank and Felleisen's reduction semantics [10, 24]. Our reduction rules are defined for a subset of SSML (focused on the novel aspects of the language), described as a lambda calculus SSMΛ whose syntax is shown in Figure 7. Figure 10 illustrates how these rules operate on a small example.

2.2.1 *Programs.*

SSMΛ programs are expressions *e* that consist of other expressions and values *v*, which include primitive functions for references, assignment, and waiting. For example, **after** d, r ← v becomes the expression **after** *d r v*, while **wait** x | y | z is written **wait** *x y z*. **par** is not encoded as a built-in because its operands are not evaluated the same way as regular arguments. Instead, it appears as an associative prefix binary operator in SSMΛ: **par** a & b & c is equivalent to both **par** (**par** *a b*) *c* and **par** *a* (**par** *b c*) in SSMΛ.

SSMΛ constructs are more primitive than those in SSML. For example let expressions **let** x = d ; b become an application of an anonymous function $(\lambda x\,.\,b)\ d$. SSMΛ includes timestamps and the unit value, but for brevity, we omit all other literals, algebraic data types, pattern matches, conditional expressions, and while loops; Pottier & Rémy [24] explain how they could be added.

e	$::=$		Expressions:
	$\mid$	v	Value
	$\mid$	$e\ e$	Application
	$\mid$	$\mathbf{par}\ e\ e$	Parallel evaluation
v	$::=$		Values:
	$\mid$	x	Named variable
	$\mid$	t	Timestamp value
	$\mid$	$()$	Unit value
	$\mid$	m	Memory location†
	$\mid$	$\lambda\, x\, .\, e$	Function
	$\mid$	$\mathbf{ref}_1$	Allocation
	$\mid$	$\mathbf{deref}_1$	Dereference
	$\mid$	$\mathbf{written}_1$	Last written time
	$\mid$	$\mathbf{after}_3$	Delayed assignment
	$\mid$	$\mathbf{assign}_2$	Instant assignment
	$\mid$	$\mathbf{wait}_+$	Wait for updates
	$\mid$	$\mathbf{check}_+$	Check for updates†
	$\mid$	$\mathbf{suspend}_+$	Suspend execution†

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

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

Values v include primitive functions such as **ref**; subscripts denote their arity (+ is one or more).

v^+ denotes one or more values.

$\mathcal{E}[e]$ is the expression produced by substituting e into the hole $\bullet$ of evaluation context $\mathcal{E}$.

Fig. 7. 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.

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

2.2.2 State: Events, the Heap and the Event Queue. The state of an SSMΛ program includes the current model time, a heap σ, and an event queue δ. Both the heap and event queue are partial maps from memory locations m to events, which are value-timestamp pairs written $v@t$. We use subscripts to extract the timestamp and value of an event, i.e., $(v@t)_\tau = t$ and $(v@t)_\nu = v$.

In both partial maps, the memory locations are the indices for SSMΛ's heap-allocated references. An event in the heap holds both the current value of its variable and information about when it was last written, used by **wait** to determine when to resume. Meanwhile, events in the event queue represent outstanding scheduled updates, where the timestamp records when the update will take place and the value stores the value to be assigned. Note that a memory location may be present in the domain of the heap but not that of the event queue; this situation happens when there is no pending update to that location.

Advancing time involves moving events from the event queue to the heap, replacing the existing event (value) of the variable on the heap. Time is advanced by the S-Tick rule, described below.

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. During this first phase, every reduction sees the same model time. In the second phase (S-Tick), time is advanced,

$$\frac{\langle e,\sigma,\delta\rangle \xrightarrow{t} \langle e',\sigma',\delta'\rangle}{\langle t,\mathcal{E}[e],\sigma,\delta\rangle \rightsquigarrow \langle t,\mathcal{E}[e'],\sigma',\delta'\rangle}\quad(\text{S-}\textsc{Reduce})$$

$$\frac{\begin{array}{c} t < t' \qquad t' \leq \min\{\delta(m)_\tau \mid m \in \mathrm{dom}(\delta)\} \\ \sigma'(m) = \begin{cases} v@t' & \text{when } \delta(m) = v@t' \\ \sigma(m) & \text{otherwise} \end{cases} \qquad \delta'(m) = \begin{cases} \textit{undefined} & \text{when } \delta(m) = v@t' \\ \delta(m) & \text{otherwise} \end{cases} \end{array}}{\langle t,s,\sigma,\delta\rangle \rightsquigarrow \langle t',[\textbf{suspend} \rightarrow \textbf{check}]s,\sigma',\delta'\rangle}\quad(\text{S-}\textsc{Tick})$$

Fig. 8. Rules for the inter-instant step relation ⇝ between configurations of the form ⟨time, program, heap, event queue⟩. S-Reduce takes a step within an instant (using reduction rules, denoted by $\xrightarrow{t}$); S-Tick advances between instants on suspended programs.

assignments scheduled for that time are made, and every point in the program that was waiting on a variable update (**suspend**) is awakened to check on its variables (**check**).

Figure 8 defines the rules for the two phases, which proceed as a series of step relations ⇝ between configurations: 4-tuples $\langle t,e,\sigma,\delta\rangle$ that consist of the current time t; the program we are evaluating e; the current values of variables, stored as events in the heap σ; and future values of variables, stored as events in the event queue δ. 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.

2.2.4 Steps within an instant. The S-Reduce runs the program in an instant by taking small steps of the form $\langle e,\sigma,\delta\rangle \xrightarrow{t} \langle e',\sigma',\delta'\rangle$, where the expression e is a tiny part of the program being evaluated (the *redex*), and both the heap σ and the event queue δ may be updated. The current (model) time, t, however, does not change during these small steps. This is the synchronous hypothesis: instructions do not advance time; only scheduled events do.

The S-Reduce rule enforces a total evaluation order through an evaluation context $\mathcal{E}$, which specifies a unique "hole" (•) where a reduction may occur within the context of a larger program. For example, the evaluation context of the form $\mathcal{E}$ 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 an applicative, left-to-right evaluation order: the function must be reduced before its argument. Similarly, the **par** $\mathcal{E}\ e$ and **par** $s\ \mathcal{E}$ forms regulate the evaluation order between **par** branches, by forcing the left branch of a **par** to be fully reduced (i.e., to a suspended program or 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 evaluation context $\mathcal{E}$.

Figure 9 lists the reduction rules used by R-Reduce within an instant:

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

R-Join terminates a **par** construct when both of its branches have terminated (were reduced to values as opposed to suspended programs). This rule returns unit, discarding the values of the two branches. The branches could have written their values to the heap or event queue.

R-Ref allocates a fresh memory location m on the heap 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.,

$$\mathrm{dom}(\sigma) = \{m \mid (m \mapsto v@t) \in \sigma\}$$

$$\langle (\lambda x.e)\ v, \sigma, \delta\rangle \xrightarrow{t} \langle [x \to v]e, \sigma, \delta\rangle \quad (\text{R-}\textsc{Beta}) \qquad \langle \mathbf{par}\ v_1\ v_2, \sigma, \delta\rangle \xrightarrow{t} \langle (), \sigma, \delta\rangle \quad (\text{R-}\textsc{Join})$$

$$\frac{m \notin \mathrm{dom}(\sigma)}{\langle \mathbf{ref}\ v, \sigma, \delta\rangle \xrightarrow{t} \langle m, [m \mapsto v@t]\sigma, \delta\rangle} \quad (\text{R-}\textsc{Ref}) \qquad \langle \mathbf{assign}\ m\ v, \sigma, \delta\rangle \xrightarrow{t} \langle (), [m \mapsto v@t]\sigma, \delta\rangle \quad (\text{R-}\textsc{Assign})$$

$$\frac{d > 0}{\langle \mathbf{after}\ d\ m\ v, \sigma, \delta\rangle \xrightarrow{t} \langle (), \sigma, [m \mapsto v@(t+d)]\delta\rangle} \quad (\text{R-}\textsc{After}) \qquad \langle \mathbf{deref}\ m, \sigma, \delta\rangle \xrightarrow{t} \langle \sigma(m)_v, \sigma, \delta\rangle \quad (\text{R-}\textsc{Deref})$$

$$\langle \mathbf{written}\ m, \sigma, \delta\rangle \xrightarrow{t} \langle \sigma(m)_\tau, \sigma, \delta\rangle \quad (\text{R-}\textsc{Written})$$

$$\langle \mathbf{wait}\ m_1 \ldots m_k, \sigma, \delta\rangle \xrightarrow{t} \langle \mathbf{suspend}\ m_1 \ldots m_k, \sigma, \delta\rangle \quad (\text{R-}\textsc{Wait})$$

$$\frac{\exists i \in \{1, \ldots, k\}, \sigma(m_i)_\tau = t}{\langle \mathbf{check}\ m_1 \ldots m_k, \sigma, \delta\rangle \xrightarrow{t} \langle (), \sigma, \delta\rangle} \quad (\text{R-}\textsc{Unblock})$$

$$\frac{\nexists i \in \{1, \ldots, k\}, \sigma(m_i)_\tau = t}{\langle \mathbf{check}\ m_1 \ldots m_k, \sigma, \delta\rangle \xrightarrow{t} \langle \mathbf{suspend}\ m_1 \ldots m_k, \sigma, \delta\rangle} \quad (\text{R-}\textsc{Block})$$

Fig. 9. Reduction rules between instantaneous configurations. R-Ref, R-Assign, R-After, R-Deref, and R-Written 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.

and to assign an event $v@t$ to location m in the heap or event queue, we define

$$([m \mapsto v@t]\sigma)(m') = \begin{cases} v@t & \text{when } m' = m; \\ \sigma(m') & \text{otherwise.} \end{cases}$$

R-Assign instantaneously assigns the value v to previously allocated memory location m.
R-After schedules an event strictly in the future, i.e., value v will be assigned to m on the heap at time $t + d > t$. Note that this operation will overwrite any pending event on m and is the only reduction rule that modifies the event queue (S-Tick updates the queue between instants).
R-Deref returns the current value of (heap) memory location m.
R-Written returns the time at which (heap) memory location m was last written.
R-Wait forces a **wait** construct to block when it is first executed (even if one of the listed memory locations has just been written) by turning it into a **suspend**, one of the choices for a suspended program s.
R-Unblock terminates a **check** construct (a rewritten **suspend**; see below) when at least one of the variables (memory location m_i) it is waiting on has been written in the current instant t.
R-Block is the opposite of R-Unblock: when none of the variables the **check** is waiting on were written in the current instant, it turns back into a **suspend** that can be awakened later.

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 express 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 7), the S-Tick rule does three things: advances time to t', moves events at time t' from the queue δ to the new heap σ', and "wakes up" all the **suspend** constructs by rewriting them to **check**. Those **suspend** 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 variables had been written when they were last reduced (R-Block).

The choice of the next time t' is what gives the sparse synchronous model its name: instead of always advancing to the "next" time step as is done in the more traditional synchronous languages, SSMΛ allows the implementation to choose any time between the current time (i.e., $t < t'$) and the time of the earliest queued event. 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 t' is what makes the event queue behave as a queue: the bound on t' is exactly the time of the soonest event in the event queue.

Once the next time t' is selected, the new heap σ' and event queue δ' are formed by removing every event in the queue at time t' and placing it in the heap, overwriting the previous event/value. Adding an event to the heap here mimics R-Assign, but only S-Tick dequeues events.

Finally, every **suspend** in the suspended program is rewritten into a **check**, which both transforms the suspended program into an expression suitable for R-Reduce and "wakes up" each of the blocked **wait** constructs.

3 COMPILING SSML

Our formal semantics specify execution in terms of a series of rewrites, but this is not a practical implementation strategy. Instead, we compile SSML to efficient C code that leverages a language runtime (Section 4) to exhibit the same behavior as the source program. In this section, we discuss our compilation scheme for SSML programs and platform-generic aspects of the runtime.

3.1 Compiling functions

Each SSML function f is compiled into two C functions: an *enter* function that allocates and initializes f's activation record and a *step* function that performs the work of f in a single instant, e.g., from when it is resumed by some event to when it suspends.

Unlike C functions, SSML functions have the ability to suspend and resume between the execution of other functions. To enable this behavior, we maintain the local state of each SSML function in a runtime-managed activation record rather than using C's native stack. Each SSML function has its own specialized activation record type that stores (**let**-bound) local variables, arguments, and other runtime data.

Each activation record starts with the generic *act_t* header shown in Figure 11 to allow the scheduler to manage them generically. Specifically, it is always safe to cast a pointer to a function-specific activation record to an *act_t* pointer. Figure 12 shows a small function we will use as a running example; Figure 13 shows the layout of its activation record.

The generic activation record header maintains information used to resume executing its suspended step function: a pointer to that step function, its control state (an encoded program counter, i.e., where to resume), and the number of its running children, so that its last child knows when to resume it. In turn, it uses the pointer to its own parent's activation record to resume the parent when it is the last child that terminates. Finally, the header contains some data used for scheduling decisions: two numbers related to its scheduling priority (described later in Section 3.5), and a Boolean indicating whether the function has been scheduled to run in the current instant.

Figure 11 also shows two helper functions used to allocate and free activation records. *f*'s enter function, shown in Figure 14, uses *enter_alloc* from Figure 11 to allocate an activation record, before populating its *f*-specific fields.

Configuration: $\langle t, e, \sigma, \delta\rangle$				**Rule Applied**
$\langle$ 10,	**let** x = **ref** 0; **par** **after** 2 x 3 ; **wait** x **wait** x ; **assign** x 4	, {}	, {} $\rangle$	R-Ref
$\langle$ 10,	**let** x = m; **par** **after** 2 x 3 ; **wait** x **wait** x ; **assign** x 4	, $\{m \mapsto 0@10\}$	, {} $\rangle$	R-Beta
$\langle$ 10,	**par** **after** 2 m 3 ; **wait** m **wait** m ; **assign** m 4	, $\{m \mapsto 0@10\}$	, {} $\rangle$	R-After
$\langle$ 10,	**par** () ; **wait** m **wait** m ; **assign** m 4	, $\{m \mapsto 0@10\}$	, $\{m \mapsto 3@12\}$ $\rangle$	R-Beta
$\langle$ 10,	**par** **wait** m **wait** m ; **assign** m 4	, $\{m \mapsto 0@10\}$	, $\{m \mapsto 3@12\}$ $\rangle$	R-Wait
$\langle$ 10,	**par** **suspend** m **wait** m ; **assign** m 4	, $\{m \mapsto 0@10\}$	, $\{m \mapsto 3@12\}$ $\rangle$	R-Wait
$\langle$ 10,	**par** **suspend** m **suspend** m ; **assign** m 4	, $\{m \mapsto 0@10\}$	, $\{m \mapsto 3@12\}$ $\rangle$	S-Tick
$\langle$ 12,	**par** **check** m **check** m ; **assign** m 4	, $\{m \mapsto 3@12\}$	, {} $\rangle$	R-Unblock
$\langle$ 12,	**par** () **check** m ; **assign** m 4	, $\{m \mapsto 3@12\}$	, {} $\rangle$	R-Unblock
$\langle$ 12,	**par** () () ; **assign** m 4	, $\{m \mapsto 3@12\}$	, {} $\rangle$	R-Beta
$\langle$ 12,	**par** () **assign** m 4	, $\{m \mapsto 3@12\}$	, {} $\rangle$	R-Assign
$\langle$ 12,	**par** () ()	, $\{m \mapsto 4@12\}$	, {} $\rangle$	R-Join
$\langle$ 12,	()	, $\{m \mapsto 4@12\}$	, {} $\rangle$	*Terminated*

Fig. 10. How the semantics operates on an example that starts at time 10 by allocating a fresh variable x with value 0, then starts two parallel tasks. The first task schedules x to become 3 after 2 time units then waits for x to be written; the second waits on x before assigning it 4. Note that once the program has suspended for the instant, the S-Tick rule applies, advances the time to the earliest queued event (at 12), writes x, and transforms each **suspend** to a **check** to wake them up. Note that each of the rules except for S-Tick operates on the configuration through the S-Reduce rule.

```
typedef struct {          // Activation record
  void (*step)(act_t *);  // Step function
  act_t      *caller;     // Where to return
  uint16_t   pc;          // Saved control state
  uint16_t   children;    // Number of children
  uint32_t   priority;    // Order in ready queue
  uint8_t    depth;       // LSB of our priority
  bool       scheduled;   // In ready queue
} act_t;
```

```
// Activation record management
act_t *enter_alloc(size_t size,
                   void (*step)(act_t *),
                   act_t *parent,
                   uint32_t prio,
                   uint8_t depth);
void leave(act_t *act, size_t size);
void activate(act_t *act);
```

Fig. 11. Generic activation record header used by the SSM runtime, alongside some API functions to help manage activation records. Each top-level function has its own activation record type struct that begins with a generic *act_t* header so the runtime can manipulate activation records generically.

```
f a =
  let loc = ref 0 ;          // Create a new reference "loc"
  wait a ;                   // Wait for a write to reference argument a
  loc ← deref a ;            // Write the value in a to loc
  par foo loc & bar a ;      // Call foo and bar in parallel, passing loc and a
  deref a                    // Return the current value of a
```

Fig. 12. An SSML function for illustrating how we compile SSML to C. This example features allocating, accessing, and assigning to references; blocking on references, and parallel function calls.

```
typedef struct {
  act_t      act;    // Common activation record header
  value_t    a;      // Argument
  value_t    loc;    // Local variable
  value_t    *ret;   // Return location
  trigger_t  trig;   // Trigger
} f_act_t;
```

Fig. 13. Activation record for the function in Figure 12, which starts with the activation record header (Figure 11)

Meanwhile, its step function, shown in Figure 15, calls *leave* before terminating, to free the record. *leave* also decrements the number of children maintained by the parent, and reschedules the parent if *f* was the last child to leave.

We heap-allocate activation records using our own memory allocator, discussed in Section 4.2.2. Others have shown that allocating activation records on the heap gives similar performance to stack allocation [2], but makes suspending and resuming processes much simpler to implement.

3.2 Representing values

```
act_t *enter_f(act_t *parent, uint32_t prio, uint8_t depth, value_t *a, value_t *ret) {
  f_act_t *act = (f_act_t *) enter_alloc(sizeof(f_act_t), step_f, parent, prio, depth);
  act->a = a;                  // Save argument
  act->ret = ret;              // Save return location
  act->trig.act = &act->act;   // Register self in trigger
  return &act->act;
}
```

Fig. 14. Enter function for *f* in Figure 12, which uses *enter_alloc* to initialize the activation record header

```
void step_f(act_t *actg) {
  f_act_t *act = (f_act_t *) actg;
  switch (act->act.pc) {
  case 0: act->loc = new_ref(pack(0));                                    // let loc = ref 0
          sensitize(act->a, &act->trig);                                  // wait a
          act->act.pc = 1; return;
  case 1: desensitize(&act->trig);
          assign(act->loc, act->act.priority, deref(act->a));            // loc <- deref a
          uint8_t depth = act->act.depth - 1;                            // par
          uint32_t prio = act->act.priority;
          dup(act->loc);
          activate(enter_foo(&act->act, prio, depth, act->loc, NULL)); // foo loc
          prio += 1 << depth;
          dup(act->a);
          activate(enter_bar(&act->act, prio, depth, act->a, NULL));   // bar a
          act->act.pc = 2; return;
  case 2: if (act->ret) *act->ret = deref(act->a);                        // deref a
          break;
  }
  drop(act->loc); drop(act->a);         // Drop references falling out of scope
  leave(&act->act, sizeof(f_act_t));    // Free activation record
}
```

Fig. 15. Step function for the function in Figure 12

```
typedef union {                  #define on_heap(v) ((v).packed_val & 1 == 0)
  uint32_t    packed_val;        #define pack(v)    (value_t) { .packed_val = ((v) << 1) | 1 }
  struct mm *heap_ptr;           #define unpack(v)  ((v).packed_val >> 1)
} value_t;
```

Fig. 16. Memory layout for the untyped runtime representation of SSML values and macros for testing and access

```
typedef uint64_t time_t; // Unboxed model time           value_t new_time(time_t init_time);
typedef struct {         // Time values                  time_t read_time(value_t time);
  uint8_t ref_count;     // Reference count
  uint8_t kind;          // = TIME_K
  time_t  time;          // Time of scheduled update
} time_obj_t;
```

Fig. 17. Representation of model time values on the heap.

```
typedef struct {           // Reference                  value_t new_ref(value_t init_val);
  uint8_t   ref_count;     // Reference count            value_t deref(value_t ref);
  uint8_t   kind;          // = REF_K                    void assign(value_t ref,
  time_t    last_written;  // When reference was written             uint32_t prio,
  time_t    later_time;    // Time of scheduled update               value_t val);
  trigger_t *triggers;     // List of sensitive processes void later(value_t ref,
  value_t   value;         // Current value                         time_t time,
  value_t   later_value;   // Scheduled value                       value_t val);
} ref_t;                                                 time_t last_written(value_t ref);
```

Fig. 18. Runtime representation and helpers for SSML references.

The *value_t* type used in *f*s activation record to store arguments and local variables (Figure 13) is our runtime representation for all SSML values. We use this uniformly-sized representation so that the rest of the runtime system can handle these values without having treat types differently.

Figure 16 shows the machine-word representation we use for values. For 32-bit processors (our main target), the bits represent either a 31-bit integer or a pointer to a larger object on the heap, which we describe in Section 4.2. On processors with 64-bit pointers, we restrict the integers to 31 bits for code portability.

The least significant bit of the machine word described by *value_t* distinguishes pointers from packed values (integers): 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. This technique is typical in functional programming languages; we based our implementation on OCaml [16].

SSML's value encoding facilitates generating polymorphic code since it allows manipulating generic SSML values (e.g., both integers and larger objects) without having to distinguish them at compile- or run-time. Figure 16 shows macros that test whether a value is a pointer to a heap object and for converting between normal C values and packed SSML values.

We store SSML model-time values on the heap since they are 64-bit integers (Figure 17). This decision avoids headaches arising from wraparound: with nanosecond precision, 64-bit wraparound only occurs once every 584 years; 32 bits afford us less than 5 seconds. Even with microsecond precision, 32-bit timestamps wrap around in a little over an hour. *new_time* allocates a new time heap object (pointed to by a *value_t*), while *read_time* reads a time value from the heap.

3.3 Scheduling references

```
typedef struct {              // Process trigger
  act_t      *act;            // Triggered process
  trigger_t *next;            // Next trigger
  trigger_t **prev_ptr;       // Back pointer
} trigger_t;
```

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

Fig. 19. Linked list of triggers, used to wake up sensitive processes when a reference is written

SSML references behave like traditional variables in imperative languages (i.e., they hold the value most recently written to them, read using **deref**), augmented with the ability to schedule a delayed assignment and for suspended routines to be reawakened by writes to a reference.

SSML allocates references on the heap using the type in Figure 18. In addition to its current *value*, references also record when they were *last_written*; these fields are read using **deref** and *last_written*. Each reference may have at most one new value scheduled for it in the future—a pending event—recorded in the *later_time* and *later_value* fields (*later_time* is *ULONG_MAX* when no event is pending). When *later_time* arrives, the *later_value* is copied to *value*, and any processes waiting on that reference (tracked by the *triggers* list) are resumed.

The *assign* function implements instantaneous assignment, and updates the *value* and *last_written* fields of a reference. In addition to the value being assigned to the reference, *assign* also takes the priority of the current routine and only schedules sensitive routines with a higher priority number to be consistent with the semantics.

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 reference that already has a pending event, the existing event is overwritten, to avoid an unbounded accumulation of events.

3.4 Suspending and resuming

A process may suspend for one of two reasons: it is blocking on an assignment to a reference (**wait**), or it is blocking on called child processes to return (a function call or **par**). In Figure 15, *f*'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 corresponding to that program counter; in most cases, this *case* immediately follows the *return*.

Before suspending, the step function conveys a wake condition to the scheduler so that it does not remain asleep forever. When a process suspends due to a **wait**, it adds itself to the trigger list of each reference it waits on. The trigger list is doubly-linked to facilitate fast removal; its node type definition is shown in Figure 19. Each node contains a pointer back to the waiting function's activation record (initialized in the enter function, such as in Figure 14), and is enqueued and dequeued using *sensitize* and *desensitize*. When a reference is written, the scheduler traverses through its trigger list and schedules any sensitive processes to execute in that instant.

A process can reuse its triggers across different **wait** suspension points, but must use a unique trigger for each reference it waits on. As such, its activation record needs to contain at least as many triggers as the maximum number of references it waits on at once. *f* from Figure 12 only ever waits on a single reference, so its activation record only needs a single trigger.

Processes also suspend when they spawn one or more child processes; they resume when all those child processes terminate. In Figure 14, we see *f*'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 15, *f*'s

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

Fig. 20. Implementation of *leave*, which deallocates an activation record and may return to its parent

step function terminates the process when control breaks out of the switch statement. As shown in Figure 20, after *leave* frees the given activation record, it decrements the *children* count of its parent. If this was the last child leaving, it resumes its parent's step function.

3.5 Function calls and priorities

SSML programs may use **par** to evaluate multiple functions in parallel, ordered according to their position in the **par** expression. For instance, the *main* function calls *foo* and *bar* in parallel at line 10 of Figure 6, with *foo* taking priority over *bar*. 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 line 2 of Figure 2. Instead, we transform these expressions to parallel function calls using an SSML 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 *sum* function from Figure 2 is translated to

```
_wait r = wait r // Lifted from sum
sum r1 r2 r =
  par _wait r1 & _wait r2 ;
  after sec 1, r ← deref r1 + deref r2
```

At a **par** call site, the parent calls each child's enter function to allocate their activation records (see Figure 14). These are passed to *activate*, which schedules those child processes for execution.

Deterministic concurrency was a key SSML design goal. 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 since most **wait** statements will stay suspended in most instants. Instead, when a reference is written, we only schedule each process that is blocked waiting on that reference.

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 thought of 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 15, the *depth* and *priority* variables and related machinery dynamically compute the new priority-depth pairs at the call site for *foo* and *bar*.

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.

4 LANGUAGE RUNTIME IMPLEMENTATION

Our language runtime consists of a scheduler for executing a program's threads during each instant, a memory manager and allocator to provide programs with safe, dynamic memory, and platform-specific code for interfacing with the program's environment. In this section, we describe the platform-agnostic scheduler and memory manager, which allow us to execute SSML programs without considering any interactions with the external environment.

4.1 The scheduler

The SSML runtime scheduler maintains two priority queues: the event queue, which holds references scheduled to be updated, ordered according to their *later_time* fields; and the ready queue, which holds activation records (functions) 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 event queue corresponds to the event queue δ from our formal semantics. Meanwhile, the ready 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 ready queue avoids "busy waiting" by maintaining only the set of processes that will unblock and run in the current instant.

The scheduler exposes a *tick* function that runs the system for an instant. It does so in two phases: performing all the reference updates queued for the instant, then running every process in the ready queue in priority order.

In the first phase, performing an event consists of removing the reference at the front of the event queue provided it is scheduled for the current instant **now**, updating its *value* and *last_written* fields, and then adding each process waiting on the reference (held in its list of triggers) to the ready 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 reference's list of triggered processes is not modified during this phase: the processes themselves are exclusively responsible for managing their triggers.

In the second phase, the process with the lowest priority number is removed from the ready 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 through a call to *activate* (which may schedule another process at the same priority) or through an *assign* call to a reference that triggers other processes at higher priorities.

The scheduler will terminate unless one of the activated functions refuses to suspend. Functions may contain unbounded recursion or loops that perform multiple iterations in a single instant, but C does not guarantee that they terminate. However, infinite loops that always eventually suspend work fine in SSML.

4.2 Memory management

Our runtime system implements automatic garbage collection based on reference counting and an allocator that mimics *malloc* and *free* in systems where an existing allocator is inadequate or unavailable.

```
struct mm {
  uint8_t ref_count;
  uint8_t kind;
  // Payload
};
```

```
enum kind {
  ADT_K,
  REF_K,
  TIME_K
};
```

```
struct adt {           // Layout of an ADT object
  uint8_t ref_count;   // Reference count
  uint8_t kind;        // = ADT_K
  uint8_t tag;         // Code of data constructor
  uint8_t fields ;     // Number of payload fields (≥ 1)
  value_t field [];    // Payload fields
};
```

Fig. 21. The layout of all heap-managed objects; the encoding of the *kind* field, and the layout of algebraic data types. The *ref_count* field holds the number of pointers to the object; the *kind* field indicates how the rest of the object should be interpreted.

```
void *alloc_page(void);
void *alloc_mem(size_t size);
void free_mem(void *p, size_t size);
```

Fig. 22. Platform-specific handlers for the memory allocator.

4.2.1 Reference-Counted Garbage Collection. The SSML runtime exposes reference counting primitives *new*, *dup*, and *drop* to allocate, duplicate, and release heap objects. Objects are initialized with a reference count of 1, which represents the number of live pointers to the object. The reference count is incremented by *dup* and decremented by *drop*. When the reference count for an object reaches 0, the object is freed, and *drop* is called on all objects it refers to. Our choice of a reference counted heap was inspired by the Perceus algorithm [25].

The reference count is maintained in the object's memory management header, shown in Figure 21. The header is placed at the start of each heap object and also contains a *kind* field that indicates how to interpret the rest of the object. When an object is freed, the memory manager uses the *kind* field to determine where and how much to scan for heap pointers to child objects.

The SSML runtime recognizes several kinds of heap objects: (user-defined) ADTs, references, and 64-bit model time values. These are enumerated in Figure 21 and dictate the interpretation of each heap object. For instance, for user-defined algebraic data types, *kind* = *ADT_K*, the *tag* field encodes the data constructor used to create the object, and *fields* indicates the number of *value_t* elements in the *field* array.

Each kind of object has its own *new* function, which calls the allocator to obtain memory for the object and initializes the *ref_count* and *kind* fields along with others depending on the kind.

The *dup* function takes a pointer to any kind of object and increases its reference count.

The *drop* function is the most complicated. It decrements the *ref_count* field and if it has reached zero, consults the *kind* field to determine what children to drop, if any, before freeing the object. For ADTs, *drop* recursively calls *drop* on each of its fields (whose number is given by *fields*). For references, *drop* calls drop on its *value* and *later_value* fields (Figure 18).

4.2.2 Allocation. To ensure responsiveness and avoid fragmentation, the SSML runtime uses an allocator that dispenses cells from one of several memory pools. Each memory pool is responsible for allocating cells up to a certain size, and maintains a free list of available cells. Memory is allocated from the smallest cell-size pool capable of accommodating the requested size. This segregated-fits allocation scheme wastes some memory to ensure that the free list can be queried in constant time.

The memory used by the SSM allocator is acquired from the operating system on demand so that the distribution of space among the memory pools can dynamically adapt to the needs of the application. To ensure platform-independence, the allocator uses three platform-specific allocation handler functions, listed in Figure 22. The allocator uses *alloc_page* to request a page of memory from the operating system. These pages can be added to memory pools on demand or pre-allocated if the allocator is given hints when it starts that indicate many impending allocations of a particular size.

The number of memory pools is configured at compile time, and remains constant throughout the execution of SSML programs. If the SSML program attempts to allocate memory larger than the largest pool cell size, the SSM allocator falls back to the system-provided *alloc_mem* handler, which must be capable of allocating arbitrarily large ranges of memory or throw a fatal error. Memory allocated by this handler is freed using the *free_mem* handler. The use of these handlers is transparent to the SSML user program.

5 INTERFACING WITH THE REAL WORLD

Like other synchronous languages, SSML prescribes temporal behavior in terms of model time, and presents the fiction of an infinitely fast processor to ensure that the semantics of real-time programs are independent of the actual platform speed. This isolation from the external environment is reflected in our runtime system's platform-agnostic core (described in Section 4): *tick* has no control over the physical time at which code is executed.

To enable meaningful execution within the real-world environment, the runtime includes platform-specific drivers to manage interactions with that environment according to each platform's own capabilities and runtime model. For instance, the driver is responsible for calling *tick* according to the passage of physical time, informed by platform-provided timers.

The driver is also responsible for punctually conveying data from and to peripheral devices via input and output references, which are arguments passed to a program's *main* function. An input handler timestamps each input event before delivering it to the runtime event queue; each output reference is given a concurrently running output process that waits for an update before emitting the value to the environment.

In reality, the fiction of an infinitely fast processor can be maintained by a "fast-enough" processor always able to perform the computation needed in each instant before the arrival of the next. However, whether a processor is "fast enough" depends on its speed, the program it is expected to execute (e.g., its worst case execution time), and the rate at which environmental inputs arrive, making it a difficult property to prove. We recognize that establishing schedulability is important, but it remains a challenging aspect of our approach; we plan to tackle this problem in future work. For the moment, our runtime system can report observed scheduling failures.

In this section, we describe the design considerations and implementation requirements common to all platforms' drivers. We have implemented a driver for the Zephyr real-time operating system, which allows us to run SSML programs on a wide range of embedded systems, along with one for POSIX, which enables us to experiment on desktop workstations.

5.1 Timing and external input

Calling *tick* advances model time (**now**) to that of the next event in the queue. We can run SSML programs "in simulation" like any discrete-event model by repeatedly calling *tick*, without regard for physical time. While this is useful for compiler testing, such simulations do not interact with the real world as prescribed by the program.

Drivers leverage platform timing capabilities to implement real-time behavior. While timer APIs differ vastly across platforms, running programs "in real-time" is still straightforward, provided some capability to sleep or block until a specified physical time (usually implemented using some sort of free-running timer).

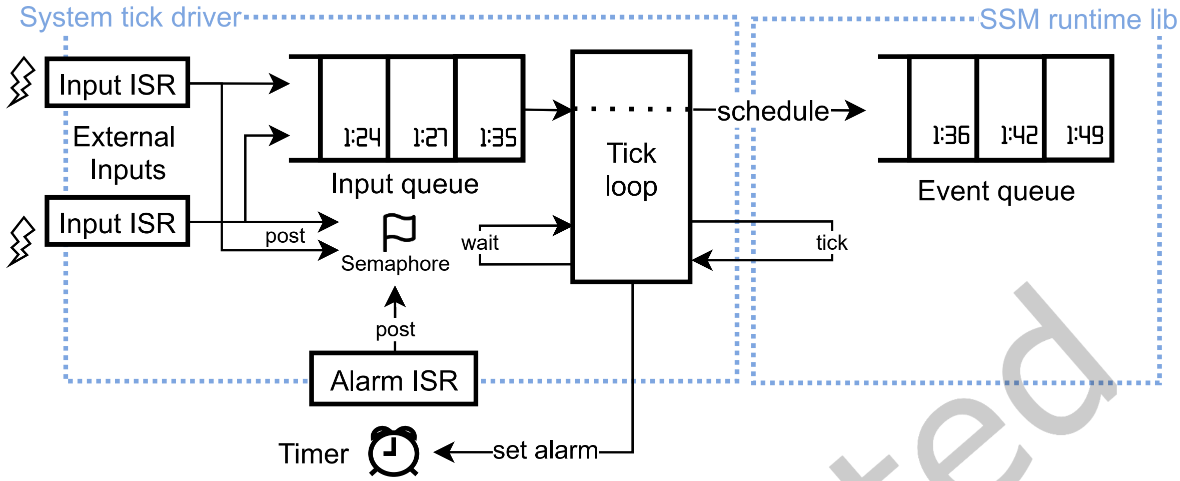

Fig. 23. The SSML runtime's input handling architecture.

External inputs are gathered by concurrent interrupt service routines (ISRs) and ultimately appear as updates to SSML input references. Handling these inputs complicates the tick loop, which contends with two challenges. First, inputs may arrive at any point: asynchronously updating the input reference in the ISR while *tick* is running may corrupt system state. Second, the tick loop must ensure that model time advances monotonically: it should only *tick* once the corresponding physical time has passed, to ensure that it has accounted for any possible inputs before that time.

Our solution adds an input queue to manage input events from the ISRs (i.e., distinct from the scheduler's event and ready queues) and a semaphore to wake up the (potentially sleeping) tick loop thread. Figure 23 illustrates this input handling architecture. An ISR gathers an input, timestamps it, places it in the input queue, and posts to the semaphore. The tick loop sleeps by waiting on this semaphore with an alarm scheduled to post when the sleep period is over. The tick loop collects input events from the input queue, forwards them to the event queue, and calls the scheduler's *tick* to run an instant.

Maintaining an input queue distinct 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 event queue is a priority queue to which events are added and sometimes removed out-of-order by the tick loop and running SSML processes. However, because these run synchronously, the scheduler event queue is never accessed asynchronously and does not need to be made thread-safe.

Figure 24 shows an input interrupt handler. It first records the system's current physical time, then attempts to enqueue an event with that timestamp and any new data from the peripheral in the input queue: a ring buffer large enough to accommodate a modest backlog of input events before having to drop new ones. Enqueue and dequeue operations are performed in-place via an allocate/commit and peek/release protocol to minimize copying.

The ISR obtains an *irq_lock* because recording the current system time as quickly as possible and allocating space in the input queue is critical to ensure enqueued input events appear with non-decreasing timestamps, at the cost of temporarily disabling nested interrupts. Higher-priority interrupts may occur while processing a lower-priority interrupt, but only after the timestamp of the lower priority interrupt has been captured and its position in the queue guaranteed.

```
struct input_event {
  time_t  time;  // Timestamp
  value_t data;  // Peripheral data
  value_t ref;   // Target reference
};
```

```
void input_isr(device *dev) {
  int key = irq_lock();
  time_t input_time = timer_read();
  struct input_event *event = input_alloc();
  irq_unlock(key);
  if (event == NULL) return;      // Queue is full!
  event->time = input_time;
  event->ref = input_binding(dev);
  event->data = input_read(dev); // Device-specific
  input_commit();                 // Enqueue input event
  sem_post(sem);                  // Wake up tick thread
}
```

Fig. 24. The type of input queue events and a peripheral interrupt service routine. Each peripheral has a device-specific ISR like this one.

```
semaphore_t sem;                                  // Operating system semaphore
void alarm_isr(void) { sem_post(sem); }           // Called when alarm goes off
void tick_loop(void) {
  init ();                                        // Initialize the SSML scheduler
  tick ();                                        // Start the program at time zero
  for (;;)  {
    time_t real_time = timer_read();              // Get physical time
    time_t mtime = next_time();                   // Get model time: the next scheduled event
    struct input_event *input = input_peek();     // Read the next queued input event, if any
    if (input && input->time <= mtime) {          // Is there also an event from the environment?
      schedule_input(input);                      // Add input to event queue
      input_release();                            // Input dequeue
    } else if (mtime <= real_time) {              // Has the model fallen behind real time?
      tick ();                                    // Run the system; advance model time
    } else if (mtime != NO_EVENT) {               // Is there a future event scheduled?
      if (timer_set_alarm(mtime, alarm_isr)) {    // Set alarm for next event. Is it in the future?
        sem_wait(sem);                            // Wait for alarm or environmental input
        timer_cancel_alarm();                     // An input event should cancel the alarm
        sem_reset(sem);                           // If timer collides with input event
      }                                           // Next event is in the past; do not sleep
    } else {                                      // Nothing pending; wait for the environment
      sem_wait(sem);                              // Wait for environmental input
} } }
```

Fig. 25. The runtime tick loop, which gathers events from the input queue, runs *tick* to advance model time, then sleeps until the next scheduled event or an interrupt.

Figure 25 shows the tick loop, which waits on an OS-provided semaphore (*sem*) that is posted by an ISR from either a peripheral or the system clock. 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 the system timer advances monotonically. Pending inputs are forwarded. However, if the model time of the SSML program is running behind physical time when some external input is received, that is, if *input−>time* > *mtime*, or if there is no pending input event, the runtime executes the SSML program by calling *tick*. Finally, if there are neither input events to process nor internal events ready to execute, the main thread goes to sleep, blocking on a semaphore until either its timer expires or some fresh input appears. Upon waking, the tick loop cancels the timer ISR and resets the semaphore to prevent stale posts on the semaphore lingering into the following loop iterations.

5.2 Handling external outputs

Just like inputs, output peripherals are bound to regular SSML references; writes to those references are sent to the environment as output. Under the hood, writes to such references trigger some system-provided function that actually transmits the output to the peripheral.

To determine when to call such output functions, we reuse our runtime's sensitivity machinery to determine when an output reference is updated. We maintain output handler processes which remain active and sensitized to the bound output reference. Those processes encapsulate communication with each output peripheral, in SSML handler functions that look like

```
handle out =
  while True do
    wait out ;
    peripheral_output  (deref out)
```

Here, *peripheral_output* represents the platform- and peripheral-specific work the output handler must do to forward the value of *out* to the environment. Adding a new peripheral amounts to defining a new output handler function according to this template. We plan to add a foreign function interface to SSML so that output handlers can be directly implemented in SSML code like this.

The runtime scheduler runs output handlers last in each instant so that they are sensitive to instantaneous assignments by all processes. In SSML, these processes may be thought of as scheduled parallel to the *main* process by some "real" entry point, *_start*, e.g.,

```
main led1 led2 =  ...

_start  led1 led2 =
  par main led1 led2
    & handle led1
    & handle led2
```

We chose to implement output handlers as low-priority processes because it ensures a sound real-world interpretation for logically simultaneous updates. A program may assign a reference multiple times in a single instant,for example, turning an LED on and off "at the same time." The zero-time model of execution means such multiple writes occur simultaneously, which is meaningless to real-world peripherals; handling each write immediately may produce flickering or glitch behavior that should not be externally visible. By only responding to the last write of any instant, our output handlers ensure that peripherals are updated at most once each instant with only its "stabilized" value.

A shortcoming of our approach is that outputs do not appear in the real-world until the end of computing an instant. SSML's semantics say that an instant executes in zero model time, but in practice any computation takes physical time. A sporadically intensive workload may lead to varying output latency, causing jitter that may be unacceptable for real-time workloads. An alternative is to schedule output handlers before the *main* process, e.g.,

```
main led1 led2 = ...

_start led1 led2 =
  par handle led1
    & handle led2
    & main led1 led2
```

Assuming that output handlers are reasonably efficient, this alternative ensures that output effects are emitted punctually without being delayed by long-running instants. However, this approach precludes effectful instantaneous assignments, which we want SSML to support for semantic consistency.

We plan to explore having the user specify how output handlers should be scheduled relative to other processes, so that users can control the trade-off between the punctuality and expressiveness of output references for their application. Because these handlers are implemented as regular processes, this endeavor requires no additional runtime support, so we are limited only by the expressiveness of our language. In some cases, it may be possible for the compiler or runtime to infer whether an output handler's priority can be safely boosted without affecting the behavior of the program.

6 EVALUATION

We tested the performance of our Zephyr-targeted runtime system by subjecting SSML programs to varying loads. We performed these experiments on a Nordic Semiconductor NRF52840-DK board, which has a 64 MHz Cortex-M4 processor, 256 kB RAM, and 1 MHz flash; it is configured to use an off-chip 16 MHz crystal oscillator as its physical time base. To produce high input loads, we connected a signal generator to the GPIO pin mapped to one of the NRF52840's buttons. We measured the output response using an oscilloscope connected to the GPIO pin mapped to one of the NRF52840's LEDs.

6.1 Frequency counter

To assess our implementation's resilience to high input load, we tested the frequency counter shown in Figure 26. This program measures the frequency of button presses by counting the number of input events each second. The reported count is double the frequency, corresponding to the two input events of a square wave. For benchmarking purposes, we inserted a print statement in the generated C code to report the frequency; the program alternates between counting and reporting to ensure that the overhead of reporting the frequency does not interfere with the frequency counting at high loads.

The reference *button* is bound to switch 0 on the NRF52840-DK; we connected a function generator to the corresponding GPIO pin and generated pulses at various frequencies. Our results are shown in Table 1. We found that our frequency counter was capable of measuring events of up to 29 kHz (input frequency of 14.5 kHz), with error within 2 Hz. Above this, the input event queue filled up faster than the counter was able to empty it, leading to events being dropped and the program thrashing. However, the frequency counter recovered gracefully from such an overload situation after we lowered the input frequency below 14 kHz.

6.2 Button-to-blink

To better understand how the system behaves without any significant computational load, we evaluated a "button-to-blink" program, shown in Figure 27. This program immediately lights an LED when a button is pressed, and turns the LED off when the button is released. We schedule the LED handler after the main *b2b* program so the instantaneous assignment is sent to the LED at the end of the instant.

```
freqCount button output =
  let count = ref 0 ;
  let gate = ref () ;
  after (sec 1), gate ← () ;
  while True do
    if written gate == now
    then
      output ← deref count ;
      count ←
        if written button == now
        then 1 else 0 ;
      after (sec 1), gate ← () ;
      wait gate; // Sleep for 1 sec
      after (sec 1), gate ← ()
    else
      count ← deref count + 1 ;
    wait gate | button
```

Fig. 26. A frequency counter program.

Input frequency	Count
6 kHz	12000 ± 0 µs
7	14000 ± 1
8	16000 ± 1
9	16000 ± 1
10	20000 ± 1
11	22000 ± 1
12	24000 ± 1
13	26000 ± 1
14	28000 ± 1
15	(events dropped)

Table 1. Frequency counter measurements. The measured count is twice the input frequency because it counts both the rising and falling edges of the input signal.

```
b2b button led =
  while True do
    wait button ;
    led ← deref button
```

Fig. 27. The button-to-blink program.

Frequency	Latency	Jitter
2 kHz	60.0 µs	0.7 µs
4	60.6	0.7
6	49.7	0.5
8	52.1	13.1
10	47.3	34.0
12	45.6	24.0

Table 2. Latency and jitter measurements on the button-to-blink program (Figure 27).

The "instantaneous" assignment in *b2b* incurs a practically avoidable but theoretically significant amount of latency, which we call the at-rest input latency, δ_r. This value represents the duration between the system waking from sleep and responding to an external input. From profiling our button-to-blink program, we found that δ_r for our NRF52840-DK is approximately 60 µs.

If the system continuously receives inputs at a frequency above $1/\delta_r$, the system will not be able to process one input before receiving the next, making δ_r a significant number. As the system falls farther behind physical time, its behavior degrades to that of an asynchronous system running as fast as it can, without the temporal behavior from the underlying SSML program.

A related metric is the in-flight input latency, δ_f, which is the time it takes for the system to respond to external input when it is already busy. When events arrive separated by less than δ_f, the input queue will be populated with new events by the time the main tick loop thread checks it again, meaning it can resume ticking without

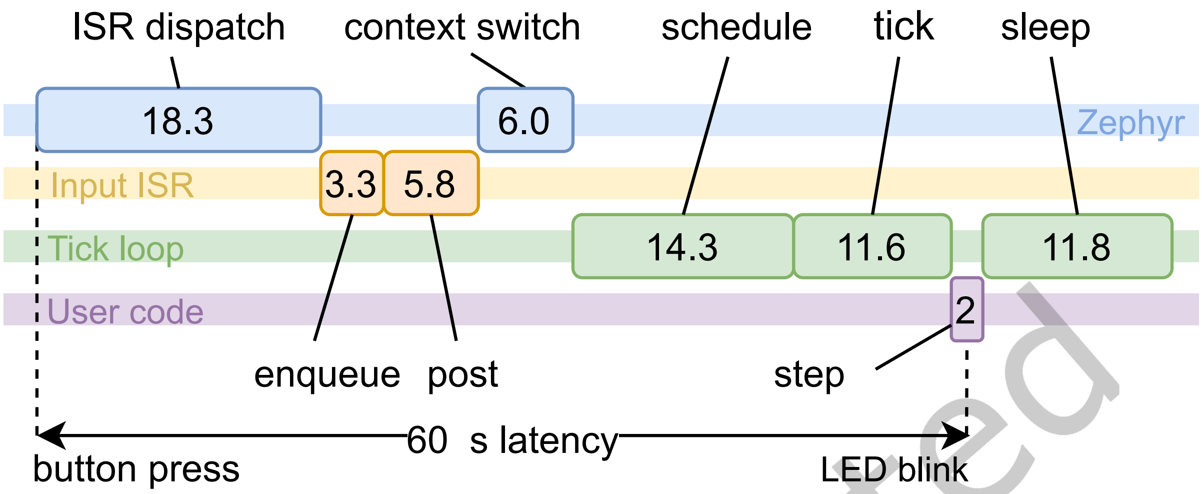

Fig. 28. Timeline of a single button press of the "button-to-blink" program from Figure 27, reconstructed from profiling codes emitted on GPIO pins. All time values are measured in µs.

putting itself to sleep. δ_f is shorter than δ_r because the time spent going to sleep and waking back up is eliminated. We experimentally determined that δ_f for our NRF52840-DK running *b2b* is approximately 45 µs.

Sparse bursts of events do not necessarily cause the system to overload, provided the system is given a chance to catch up between bursts. The system may even be able to keep up if it is consistently stimulated with a period between δ_r and δ_f. As shown in Table 2, the "button-to-blink" program was able to sustain activity when stimulated by a pulse generator with frequencies of 12 kHz; beyond 12 kHz, the program begins to thrash and drop input events. However, when the delay between successive events is less than δ_r, the computation time per instant becomes less predictable, even as the system remains responsive. In Table 2, this manifests in increased jitter at input frequencies above 8 kHz, at which a square wave has a half-period of 62.5 µs.

To determine the breakdown of our system's δ_r of 60 µs, we profiled *b2b* during a single button press and obtained the timeline shown in Figure 28. We did so by emitting 4-bit codes on unused GPIO pins at certain points in the execution of our runtime system and recording them with a logic analyzer. We then analyzed the recorded data to determine how long the system is spending between these events. Although each GPIO write takes 60 ns, this duration is negligible compared to other latencies. We find that of the overall latency, 24.3 µs is due to Zephyr's own ISR and process scheduling facilities. A further 5.8 µs is introduced by Zephyr's semaphore implementation, when we call *sem_post* from the ISR to wake up the tick loop thread. The δ_r of 60 µs measured here is consistent with the timing recorded in Table 2.

6.3 Signal generator

We also measured the highest frequency (shortest period) a signal generator program such as Figure 1 could generate. For the purposes of evaluation, we used a modified version that adjusted the half-period *hperiod* in linear increments of 2 µs. We found that it could reliably generate signals with a 76 µs half-period (approaching δ_f), corresponding to a frequency of about 6.6 kHz. Even at this frequency, our signal generator exhibited less than 500 ns of jitter, the minimum we could measure with our oscilloscope. At half-periods between 70 µs–76 µs, we observed significant degradation in signal accuracy and consistency: the output signal's half-period fluctuated between 60.2 µs and 83.4 µs. At even shorter half-periods, the system ceased to output any signal while it was computationally overwhelmed. However, the signal generator was able to recover without a reset when we increased the half-period back above the 76 µs threshold: the system would output a brief, high-frequency burst as it caught up with physical time, before resuming correct behavior.

7 RELATED WORK

7.1 Discrete-event languages

The Lingua Franca (LF) coordination language [17, 18] has many parallels with SSML. LF is inspired by the same foundations of discrete-event simulation [15] and the Ptides programming model [31, 32] as SSML; its execution model also uses two priority queues to schedule pending events and computation, sparing *tick* the need to run in every instant. In LF, locally stateful "reactors" (like SSML processes) send each other discrete, timestamped events via "ports" (like SSML channels).

The key difference 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 and insisting on minimum reactor delay times to reject causality violations. By contrast, SSML allows processes to spawn other processes at runtime, forming a dynamic process tree where the execution order is computed using the priorities obtained from parallel function call sites.

Verilog [13] and VHDL [28] inspired some aspects of SSM, while also warning us of nondeterministic pitfalls. Both are imperative discrete-event simulation languages for modeling digital hardware, and use variables that convey events (signals in VHDL; nets and regs in Verilog). SSML's assignment and **after** parallel Verilog's blocking and non-blocking assignments.

VHDL exposes far more of the discrete-event machinery to the user, e.g., allowing her to control the filtering of closely spaced events (transport vs. inertial delay), test for the presence of events, and even check for the absence of events over a prescribed period of time.

We wanted SSML to have the same power as VHDL's **wait**, which can wait for three things: an event on a signal (like SSML); a condition (e.g., *wait until CLK'event and CLK='1'*); and a period of time (e.g., *wait for 10 ns*). This felt too rich for an SSML primitive, so we adopted the approach of the FreeHDL compiler [22], whose runtime can only wait on a set of signals. FreeHDL implements VHDL's **wait** for conditions by generating code that alternates between checking the condition and waiting for an event on any of the condition's input signals. Waiting for a timeout schedules an event on a (synthesized) timeout signal then waits on that signal (Figure 5).

A key advantage of SSM is its use of a separate semantics for dealing with events in a single instant, unlike traditional discrete-event models. SSM could be modeled with *superdense* time [14] where the timestamp at which each process executes also includes its priority. This approach gives SSM its determinism and sidesteps such infelicities as VHDL's delta cycles [28].

7.2 Synchronous languages

SSM's intra-instant semantics are rooted in the synchronous languages Lustre, Esterel, and Signal, which gave a formal foundation for the semantics of synchronous computing with deterministic concurrency [4]. These languages do not support dynamic process creation, runtime scheduling, or recursion, which allows each program to be compiled to a single tick function that evaluates the whole system for an instant. The runtime simply calls this tick function in a periodic loop. This "heartbeat" model works well for continuously evolving systems, but ticks unnecessarily for reactive applications with sparse, irregular workloads. SSM assumes that events are sparse, so its runtime only calls *tick* when needed; in most instants, no computation takes place.

While Lustre and Signal are dataflow languages, Esterel is imperative like SSML: Esterel programs describe processes by their control flow [5]. Yet Esterel institutes a single-value-per-instant rule for signal values (signal coherence), and insists that all readers of a signal execute after writers, so the execution order of parallel processes is determined by their implicit *causality*. Yet causality analysis is difficult to explain and implement [23], and rejects programs with read-modify-write behavior (e.g., Figure 6). Sequentially constructive concurrency [27, 30] 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. SSML uses a simpler syntactic total order from **par** expressions, foregoing the need for causality analyses.

Reactive C takes a different approach to maintain causality while avoiding the overhead of causality analysis [7]. Reactions to the *absence* of signals are delayed by one instant, ensuring causality by construction. ReactiveML builds on this insight to incorporate synchronicity and valued signals in an ML-like language with first-class functions and ADTs [19, 20]. ReactiveML also delays processes waiting on valued signals by one instant. In contrast, SSML's sparse model of time lacks the notion of a "next instant," and its semantics do not insist on this notion of causality. In SSML, processes may check whether a reference *r* has been assigned in the current instant using **written** *r* == **now**.

We were also inspired by Boussinot's ordering of concurrent routines to achieve determinism, at the heart of FairThreads [8] and FunLoft [9]. However, Boussinot's execution strategy searches for Esterel-like fixed-points, using a round-robin cooperative scheduler that repeatedly evaluates concurrent routines in order until they quiesce. For example, in SSML, **par** *foo & bar* runs *foo* then *bar* once each in an instant; Boussinot would also run *foo* then *bar*, but allows *foo* and *bar* to resume (e.g., if one routine wrote to a variable on which the other was blocked). This repeats until each routine either terminates or suspends on an untouched variable. Boussinot's iterations enable instantaneous bidirectional communication among processes, but their execution time is difficult to bound. Also, confusingly, the lexicographic order of concurrent routines still matters in Boussinot's world. SSML adopts a more rigid, faster policy that is simpler to explain.

Hanxleden, Bourke, Girault, and others [6, 29] show how giving an Esterel program the ability to schedule when it should be awakened after the end of each tick enables far richer temporal behavior. Their proposal, however, leaves the subtle calculation of this single number to the program itself. Their solution [29] effectively implements a crude event queue in Esterel where, at each tick, each pending delay action reports its desired wake-up time to a global signal that computes the earliest event and reports that to an external timer. SSML uses a much more efficient priority queue that avoids each delay having to do something at each tick.

Both Céu and PRET-C employ many of the same ideas as SSML to overcome Esterel's complexity [1, 26]. Both languages determine the execution order of parallel branches by their syntactic order, though Céu discourages the use of non-commutative **par** branches. According to this order, earlier writes are overwritten by subsequent writes in the same instant, like SSML. Both also empower the programmer to program in terms of physical time: Céu supports blocking for a concrete duration (e.g., **await** *2ms*), while PRET-C leverages worst-case reaction time analysis and precision-timed hardware guarantees to tell the programmer the physical duration of each logical tick. However, Céu and PRET-C are statically scheduled and memory-bounded; though these are attractive properties for real-time applications, they also come at the cost of language expressiveness. SSML strikes a different balance between efficiency and expressive power to support languages features such as recursion and heap allocation.

8 FUTURE WORK AND CONCLUSIONS

We presented the Sparse Synchronous Model via a language with parallel function calls and blocking waits on writes to shared references, which may be scheduled in the future to provide temporal control. We discussed the semantics of our model and presented an efficient runtime system with two priority queues: one for events, and the other for ready-to-run routines. The result is a deterministic formalism that supports precise timing specification, concurrency, and recursion.

Our end goal is a user-friendly language built on SSM's semantics; the SSML language we present here is a step toward that. We are extending SSML into a richer functional language with arrays, first-class functions, and type classes. A foreign function interface will be useful for supporting peripherals in the embedded applications we will write; robust compile-time facilities and domain-specific optimizations will help our programs run efficiently on low-powered hardware.

We will continue to improve our runtime's reliability and responsiveness. Our current tick loop only advances model time once the corresponding physical time has passed, in case of then-unknown real-time inputs that may interrupt the passage of model time. This conservative approach means that each instant is always completed slightly late. We intend to experiment with distributed implementation techniques proposed for Ptides, and execute instants early when it is safe to do so [33]. Our use of the *depth* value limits the call depth of **par** expressions with two or more operands; to overcome this limitation, we plan to implement Dietz and Sleator's list-range relabeling algorithm [3, 11] as a solution to the order-maintenance problem. We hope to statically determine when we can relax SSM's strict child-ordering rules without introducing nondeterminism—perhaps using Rust-like ownership types [21]—to enable parallelism.

The temporal semantics specified by SSM programs mean that they convey a very precise model of when a system is active or idle. In particular, idle periods are abundant for the sparse workloads we envision. We believe our memory manager can exploit this knowledge and defer certain routines to idle cycles. For example, dropping the last remaining reference to a linked list entails iterating through the entire list and dropping every single node. When the linked list is long, doing so wastes precious cycles when there may be more urgent tasks in that instant. A lazier approach that defers memory management to idle periods may lead to better responsiveness, without much of the guesswork involved in traditional, non-synchronous memory management.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their thorough review and insightful feedback. Robert Krook, Bo Joel Svensson, and Koen Claessen developed an EDSL and property-based testing infrastructure to field-test our language runtime. Kent J. Hall integrated our build system with Zephyr's; Daniel Scanteianu wrote the first draft of the memory allocator; Hans J. Montero wrote runtime bindings for the POSIX platform; Xijiao Li provided insights toward the formulation of *SSMΛ*'s reduction semantics.

REFERENCES

[1] Sidharta Andalam, Partha S. Roop, and Alain Girault. 2010. Predictable multithreading of embedded applications using PRET-C. *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)* (2010), 159–168.

[2] Andrew W. Appel and Zhong Shao. 1994. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming* 6 (1994).

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

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (Jan. 2003), 64–83.

[5] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (Nov. 1992), 87–152.

[6] Timothy Bourke and Arcot Sowmya. 2009. Delays in Esterel. In *Proceedings of SYNCHRON*. Schloss Dagstuhl, Germany. Dagstuhl Seminar 09481.

[7] Frédéric Boussinot. 1991. Reactive C: An Extension of C to Program Reactive Systems. *Software: Practice and Experience* 21, 4 (April 1991), 401–428.

[8] Frédéric Boussinot. 2006. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience* 18, 5 (April 2006), 445–469. https://doi.org/10.1002/cpe.919

[9] Frédéric Boussinot. 2010. *Safe Reactive Programming: The FunLoft Language.* Lambert Academic Publishing.

[10] Erik Crank and Matthias Felleisen. 1991. Parameter-passing and the lambda calculus. In *Proceedings of Principles of Programming Languages (POPL).* ACM Press. https://doi.org/10.1145/99583.99616

[11] P. Dietz and D. Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87).* Association for Computing Machinery, New York, NY, USA, 365–372. https://doi.org/10.1145/28395.28434

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

[13] IEEE Computer Society 1996. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364–1995).* IEEE Computer Society, New York, New York.

[14] Edward Lee and Haiyang Zheng. 2005. Operational Semantics of Hybrid Systems. *Lecture Notes in Computer Science* 3414, 25–53. https://doi.org/10.1007/978-3-540-31954-2_2

[15] Edward A. Lee. 1999. Modeling Concurrent Real-Time Processes using Discrete Events. *Annals of Software Engineering* 7 (1999), 25–45.

[16] Xavier Leroy. 2002. Compiling Functional Languages. Presentation at Spring School "Semantics of Programming Languages". https://xavierleroy.org/talks/compilation-agay.pdf

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

[18] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2020. Reactors: A Deterministic Model for Composable Reactive Systems. In *Cyber Physical Systems. Model-Based Design*, Roger Chamberlain, Martin Edin Grimheden, and Walid Taha (Eds.). Springer International Publishing, Cham, 59–85.

[19] Louis Mandel, Cédric Pasteur, and Marc Pouzet. 2015. ReactiveML, Ten Years Later. In *Proceedings of 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming.* Siena, Italy. https://doi.org/10.1145/2790449.2790509

[20] Louis Mandel and Marc Pouzet. 2005. ReactiveML, a Reactive Extension to ML. In *Proceedings of 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming.* Lisbon, Portugal. https://doi.org/10.1145/1069774.1069782

[21] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. *ACM SIGAda Ada Letters* 34, 3 (Nov. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[22] Edwin Naroska. 1998. The FreeHDL Compiler/Simulator System. Online at http://freehdl.seul.org/.

[23] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel.* Springer. http://www.springer.com/prod/b/978-0-387-70626-9

[24] Franois Pottier and Didier Rémy. 2004. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Cambridge, Massachusetts, Chapter 10.

[25] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM. https://doi.org/10.1145/3453483.3454032

[26] Rodrigo C. M. Santos, Guilherme F. Lima, Francisco Sant 'Anna, Roberto Ierusalimschy, and Edward H. Haeusler. 2018. 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.* ACM. https://doi.org/10.1145/3211332.3211334

[27] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. 2019. Practical Causality Handling for Synchronous Languages. In *Proceedings of Design, Automation, and Test in Europe (DATE).* Florence, Italy, 1281–1284. https://doi.org/10.23919/DATE.2019.8715081

[28] The Institute of Electrical and Electronics Engineers (IEEE) 1988. *IEEE Standard VHDL Reference Manual (1076–1987).* The Institute of Electrical and Electronics Engineers (IEEE), New York, New York.

[29] Reinhard von Hanxleden, Timothy Bourke, and Alain Girault. 2017. Real-Time Ticks for Synchronous Programming. In *Forum on Specification and Design Languages (FDL).* Verona, Italy. https://doi.org/10.1109/FDL.2017.8303893

[30] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. 2014. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. *ACM Transactions on Embedded Computing Systems* 13, 4s (July 2014), 144:1–144:26.

[31] Yang Zhao, Jie Liu, and Edward A. Lee. 2007. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *Proceedings of Real-Time Technology and Applications Symposium (RTAS).* 259–268. https://doi.org/10.1109/RTAS.2007.5

[32] Jia Zou. 2011. *From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems.* Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[33] Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng, and Patricia Derler. 2009. Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems. In *Proceedings of Real-Time Technology and Applications Symposium (RTAS)*. San Francisco, California, 77–86. https://doi.org/10.1109/RTAS.2009.39