The Sparse Synchronous Model

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, USA
ORCID 0000-0003-2609-4861

John Hui
Department of Computer Science
Columbia University
New York, USA
ORCID 0000-0002-6355-3767

***Abstract*—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 present SSM, its motivations, and details of a lightweight runtime system upon which a future language will be built.**

***Index Terms*—real time systems, concurrency control, computer languages, timing**

I. Introduction

Real-time data collection is one motivation for this work, which aims to enable programmers to specify temporal software behavior as precisely as its function. 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 C on a microcontroller with timers requires a sophisticated programmer (e.g., not the typical biologist) and is difficult to reproduce. Our colleague had moved to a microcontroller running a periodic cyclic executive that simulated a finite state machine stored in an array, but found the FSM model limiting and the timing precision insufficient.

Our Sparse Synchronous Model (SSM) addresses these needs. Rats are not periodic enough for an RTOS and the synchronous languages [1] generally employ the heartbeat model. Ptides [2] avoids a heartbeat but is awkward at sequencing. Dynamic Ticks [3] comes closer to our goals by adding a low-level "wake-up call" facility to Esterel. See Section V.

Our goals for SSM were precise (μs-level) timing specification and measurement, deterministic I/O independent of platform speed, concurrency, computational efficiency without a heartbeat, recursive function calls, and bounded memory. The system uses bounded memory if it has bounded recursion, but the remaining properties are provided by the model itself. We know of no other that provides all of these together.

Our primary goal was precise specification and measurement of real-time events. Hence, our model treats time as a first-class object, like a discrete-event simulator. Inspired by the synchronous languages [1] and Ptides [2], an SSM system operates with *model time*, which a sufficiently fast implementation keeps synchronized to wall-clock time. SSM is defined on and deterministic with respect to model time, in which computation is instantaneous and deadlines are never missed. A particular implementation might miss some deadlines, but doing so will not affect its state (the environment might react) and running the system on a faster platform will bring its behavior strictly closer to ideal.

Model time is discrete to preclude Zeno-like behavior, although for platform-independence, the fundamental time quantum is not visible. An SSM system only sees time in seconds. We envision implementations with 1μs precision.

SSM solves the challenge of providing deterministic concurrency by totally ordering the execution of concurrent tasks within an instant. Specifically, it runs tasks using cooperative multitasking in a programmer-prescribed order. Discrete-event models such as what SSM adopts between instants are usually nondeterministic because they erroneously treat simultaneous events (with identical timestamps) as order-independent. While prohibiting truly simultaneous events might seem an attractive solution, they appear inherent to concurrent systems.

For both timing precision and efficiency, we wanted a model that did not rely on a "heartbeat" approach in which the system must march in lockstep with a periodic clock such as a 10 ms kernel interrupt. Instead, the SSM runtime system performs some work in response to an event and then schedules itself to be awakened by a precision hardware timer or another event

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. Besides two fixed-length arrays used to manage priority queues of pointers into activation records, the SSM runtime uses no additional run-time data beyond these activation records, so if the maximum number of activation records can be determined at compile-time, an SSM system runs in bounded memory, a useful trait for our target platforms of resource-constrained microcontrollers.

We present SSM in three sections: through a toy language (Section II), its semantics (Section III), and a C implementation of its runtime (Section IV).

This work was supported in part by the National Institutes of Health (NIH) under grant 1RF1MH120034-01.

```
program ::= routine*
routine ::= routine-id ( [ | arg [, arg]* ] ) vdecl* stmt*
arg     ::= var-id | & var-id Pass-by-value and -by-reference
vdecl   ::= var var-id = expr          Local variable declaration
expr    ::= var-id | @ var-id | literal | expr [ + | − | * | < ] expr
stmt    ::= var-id = expr                   Immediate assignment
          | if expr then stmt* [ | else stmt* ]      Conditional
          | while expr stmt*                           Iteration
          | after expr s var-id = expr       Delayed assignment
          | wait var-id [ var-id ]*                   Suspension
          | fork [ routine-id ( [ | expr [ , expr ]* ] ) ]*   Call
```

Fig. 1. Abstract syntax of SSM systems. Brackets [], bars |, and asterisks * denote syntactic grouping, choice, and zero-or-more. Tokens are bold.

II. Informal Presentation

Fig. 1 is the abstract syntax of a toy language illustrating SSM systems. For clarity, it only manipulates integers, and the only novelty in the expression syntax is the @ operator, which reports whether its variable was written in this instant.

An SSM program (e.g., Fig. 2), consists of *routines* that may call each other concurrently and recursively over multiple instants. We call them routines because they are coroutine-like and are run for their side effects, unlike functions. Each routine consists of imperative statements including variable assignment, conditionals, and loops. Arguments to routines may be passed by value or by reference (denoted by &), which enables routines to return values and to communicate among themselves. The system starts at the *main* routine, which may in turn run others. An imperative program expressed in SSM behaves in the usual way.

SSM assumes an infinitely fast processor to make its behavior as platform-independent as possible. Specifically, model time only advances for *wait* and *fork* statements; all other statements terminate in the instant they were started.

While variables in SSM can be used like variables in a traditional imperative language, they can also be used as communication channels. Like condition variables in Pthreads [5], variables in SSM announce when they are written, which can wake up "threads" (routines) that suspended to wait on them. However, unlike Pthreads, computation in SSM is totally ordered, and therefore deterministic; there are no data races.

An SSM system communicates with its environment through variables passed by reference to the *main* routine. Incoming data from the environment appears in input variables as events whose timestamps indicate when, in physical time, the data appeared. Similarly, scheduling an event on an output variable causes the runtime system to transmit the data at the prescribed time. Once the runtime environment is configured to communicate via these input/output variables, SSM may be used to build real-time reactive components.

SSM has three additional statements that provide temporal control and concurrency: *after*, which schedules a future variable update; *wait*, which suspends the execution of a routine until at least one of a set of variables is written to; and *fork*, which starts the concurrent execution of child routines.

```
 1 mywait(&r)
 2   wait r            // Suspend then wait for a write on r
 3 sum(&r1, &r2, &r)
 4   fork mywait(r1) mywait(r2) // Wait for r1 and r2
 5   after 1 s r = r1 + r2     // Return the sum after 1 second
 6 fib(n, &r)
 7   var r1 = 0                // Result for fib(n-1)
 8   var r2 = 0                // Result for fib(n-2)
 9   if n < 2 then
10     after 1 s r = 1  // Base case, assigned after 1 second
11   else              // Recurse: r = fib(n−1) + fib(n−2)
12     fork fib(n−1, r1) fib(n−2, r2) sum(r1, r2, r)
13 main()
14   var r = 0
15   fork fib(13, r)
```

Fig. 2. A contrived Fibonacci example in SSM with delayed assignment (*after*), waiting for variable writes (*wait*), and concurrent routine calls (*fork*)

The *after* statement schedules a particular variable to be assigned in a later instant and terminates instantly. In Fig. 2, both *after* statements (in lines 5 and 10) delay one second before the new value of r is assigned; in general the time may be given by an expression. Time delays may not be zero (normal assignment statements are used for this) or negative.

SSM only allows one outstanding update per variable—subsequent updates to variables overwrite previous updates. This avoids an unbounded accumulation of updates, and resolves nondeterminism arising from instants with multiple scheduled updates.

A *wait* statement causes a routine to suspend its execution in the current instant, and reawaken in the next instant in which any of a set of variables has been written. It is the only statement to directly advance time (*fork* statements indirectly incur time delays when the routines they call execute *wait* statements). In Fig. 2, the *mywait* routine only waits on a single variable; a statement such as *wait a b c* awakens when at least one of variables a, b, or c is written to. Afterwards, the *written* operator (@) can be used to check which of the variables was written. As demonstrated in Fig. 3 on line 4, this can be used to implement *timeout2*, which acts like a *wait* statement with two arguments, except it will also resume when the timeout passed as its first argument expires.

Unlike discrete-event languages like VHDL [6] designed for digital logic simulation, SSM routines are awakened by *any* write to a variable, not just writes that change the variable's value. We chose the event-on-write policy because we wanted to make event generation explicit. Our policy enables us to model pure events through variables that only take a single value ("unit") and to allow variables to convey sequences of values without two identical values in sequence being

```
timeout2(t, &a, &b)
  var tt = 0
  after t s tt = 0
  wait a b tt

main()
  var a = 4
  var b = 3
  fork timeout2(3, a, b)
```

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

```
1  foo(&a)
2    wait a
3    a = a * 2 // Runs after bar

4  bar(&a)
5    wait a
6    a = a + 4 // Runs first

7  main()
8    var a = 0
9    after 1 s a = 1
10   fork bar(a) foo(a) // bar will run before foo
11   // a = 10 here
```

Fig. 4. 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.

inadvertently merged. For example, the *tt* variable used for indicating a timeout in Fig. 3 conveys a pure event. SSM can also express VHDL's event-on-change policy by enclosing each assignment in a conditional that only writes to a variable if its proposed new value differs from its existing one.

The *fork* statement performs concurrent, recursive routine calls, and are SSM's sole mechanism for invoking routines (expressions may not include function calls). While it resembles a traditional function call, a fork may also indirectly advance time when *wait* statements within the called routines block. When a *fork* executes, the calling routine suspends its execution, and each of the called child routines runs either to completion, or until it suspends at a *wait*. A *fork* may terminate instantaneously if all its children do, or may incur time if any of its children suspend. In Fig. 2, the first *fork* (line 4) spawns two copies of the *mywait* routine to wait for variables *r1* and *r2*, and only terminates once both variables have been written to. The second *fork* (line 12) spawns two recursive copies of *fib* along with a copy of *sum*, which waits for the two results to be produced, sums them, and writes the sum to *r* after 1 second. Routines do not implicitly return values; a child may return a value to a parent by writing to a pass-by-reference argument, such as the various *r* variables in Fig. 2.

Fork statements are conjunctive—they wait until *all* operands have terminated—while *wait* statements are disjunctive—they resume when *any* of their operands are written. This distinction explains why we use two calls of *mywait* in Fig. 2 (line 4): *wait r1 r2* would have terminated after only the first of *r1* and *r2* had arrived, whereas *sum* needs the new values of both *r1* and *r2*. In this implementation, *fib(n−2, r2)* terminates 1 second sooner than *fib(n−1, r1)* for *n* greater than 3.

To ensure determinism within each instant, the children of a *fork* are executed in the order they are listed. Fig. 4 illustrates this: at 1 second, the delayed assignment to *a* will wake up both *foo* and *bar*. However, because *bar* appears before *foo* in the *fork* statement on line 10, *bar* will run first, reading the new value of *a* (1) and changing it to 5, then *foo* will run, multiplying *a* by 2 to produce 10.

III. Semantics

We present the semantics of SSM more formally. Below, time is a natural number (including 0), denoted by $\mathbb{N}$, and Σ is the set of data values. These semantics treat both abstractly.

Let $\mathscr{R} = \mathbb{N}^*$ be the set of *pathnames* for running routines, which are named hierarchically because *fork* imposes a tree structure. The *main* routine is denoted by the empty path (); (0) is its first forked child; (1) is its second forked child; (0;1) is the second forked child of the first forked child, etc. These names function like stack pointer values. We write ; for appending a child, e.g., if $r = (2;3)$ and $c = 5$, $r;c = (2;3;5)$.

Let $\mathscr{P}$ be the set of program counter values, i.e., that refer to a specific instruction in a routine, and let $\mathscr{C} = \mathscr{R} \to \mathscr{P}$ be partial functions that hold the control state of running routines.

Let $\mathscr{V}$ be the set of all variable names (e.g., strings), $\mathscr{A} = \mathscr{R} \times \mathscr{V}$ be the set of all *memory addresses*, and $\mathscr{S} = \mathscr{A} \to \Sigma$ be partial functions that represent each variable's current value.

Because the variables available in a routine may be a mix of local variables and those passed in by reference, we introduce the notion of an environment $\mathscr{M} = \mathscr{R} \to \mathscr{V} \to \mathscr{A}$ that, given a pathname to a running routine, returns a function that indicates the address of every local variable. For local variables and pass-by-value arguments, the environment simply returns address of the variable in the routine's activation record. For pass-by-reference variables it returns the address of the variable in the routine in which the variable was originally defined. That is, pass-by-reference variables are aliased. For example, if variable v in routine r is passed an argument v' to a child r' in environment M, $M(r(v)) = M(r'(v'))$.

Let $\mathscr{E} = \mathbb{N} \times \mathscr{A} \times \Sigma$ be the set of *events*; $(t,a,n) \in \mathscr{E}$ indicates that at time t, address a is to be updated to n.

Between instants, the state of a program is represented by a 4-tuple $(C,\sigma,M,E) \in \mathscr{C} \times \mathscr{S} \times \mathscr{M} \times 2^{\mathscr{E}}$, representing the control state of the program, the stored value of each variable, the variable name environment of each running routine, and the set of scheduled events.

A. Execution in an instant

During an instant, a set $W \subseteq \mathscr{A}$ is maintained that holds the addresses of every variable that was written to in the instant. W begins empty in every instant, i.e. $W = \emptyset$.

The execution of the program in state (C,σ,M,E) at time t updates this state in two steps. In the first step, every event in E at time t is applied, i.e., if $(t,a,n) \in E$, σ is updated so that $\sigma(a) = n$ and a is added to W. Then, in the second step, the program resumes from the *main* routine (pathname $r = ()$), which in turn resumes its children.

A routine $r \in \mathscr{R}$ can suspend at either a *wait* or a *fork* statement, so when resuming r, we must consider both:

- *wait* $v_1\ v_2 \ldots v_k$ There are two cases: If at least one of its awaited variables $v_i \in W$ has been written in the current instant, then $C(r)$ is advanced to the statement following the *wait* and the routine is executed (see below). Otherwise, the routine immediately continues its suspension without updating its control state.
- *fork* Each child is resumed in order.

Executing a routine $r \in \mathscr{R}$ depends on the statement at $C(r)$:

- $v = n$ The expression is evaluated to give value n (which may use $M(r)$ to retrieve the values of local variables in σ and may look in W to determine whether a variable has been written for an @ expression), the store is updated, i.e., $\sigma(M(r(v))) = n$, $M(r(v))$ is added to W, the set of written variables ($M(r(v))$ is the "address" of variable v in routine r), the control state $C(r)$ is updated to the next instruction, and r is executed again.
- *var* $v = n$ Variable declarations behave like immediate assignments; no variable may be left uninitialized.
- *if-then-else*; *while* These behave classically: evaluating the conditional expression, updating the routine's control state $C(r)$ accordingly, and executing r again.
- *after* d **s** $v = n$ The expressions d and n are evaluated; any existing event for $M(r(v))$ is removed from E and $(t+d, M(r(v)), n)$ is added to E; the control state $C(r)$ is updated to the next instruction; r is executed again.
- *wait* The routine suspends here; no further instructions in the routine are executed in the current instant.
- *fork* First, the arguments to all the children are evaluated. Next, argument values are sent to the children: if a pass-by-value argument named v in the cth child is being passed some value n, it is given the address $a = (r;c,v) \in \mathscr{A}$, the child's environment is updated so $M(r;c)(v) = a$, and the store is updated so $\sigma(a) = n$; if v were a pass-by-value argument being passed variable v', the child's environment is updated to point to v', i.e., $M(r;c)(v) = M(r)(v')$. Next, each local variable v in child c is added to the environment, i.e., $M(r;c)(v) = (r;c,v)$. Next, the control state of each child, i.e., $C(r;c)$, for $c = 0,1,\ldots k$, is set to the beginning of each child's routine. Finally, each child (whose paths are $(r;0)$, $(r;1)$, $\ldots$, $(r;k)$) is executed in sequence.
- When control "falls off the end" of a routine, its control state $C(r)$ is forgotten (to mark it as terminated), its caller p is identified (by truncating its pathname), and if all of p's children have also terminated (i.e., $(p;0)$, $(p;1)$, $\ldots$, $(p;k)$ are all undefined), then $C(p)$ is advanced to the statement after the *fork* that spawned this child and p is executed.

An SSM system's state is unchanged in instants where there are no scheduled events. When the system resumes, $W = \emptyset$ because there are no events; thus all children suspended at *wait* statements immediately suspend again because no variables have been written.

```
// Routine activation record management
rar_t *enter(size_t size, void (*step)(rar_t *), rar_t *caller,
             uint32_t priority, uint8_t depth)
void call(rar_t *rar)
void fork(rar_t *rar)
void leave(rar_t *rar, size_t size)

// Variable management
void initialize_type(cv_type_t *var, type val)
void assign_type(cv_type_t *var, uint32_t priority, type val)
void later_type(cv_type_t *var, uint64_t time, type val)
bool event_on(cv_t *var)

// Trigger management
void sensitize(cv_t *var, trigger_t *trigger)
void desensitize(trigger_t *trigger)
```

Fig. 5. API functions used in C functions that implement SSM routines. *enter* and *leave* allocate and free generic activation records; *call* runs a single child; and *fork* starts one of many children. For a supported type of variable such as *int*, *initialize_int* initializes its fields; *assign_int* performs an immediate assignment, which may schedule routines suspended on the variable; *later_int* schedules a future update to the variable; *event_on* reports whether a variable has been written in the current instant, either by a scheduled event or an immediate assignment. *sensitize* adds a trigger for the routine to the given variable; *desensitize* removes it.

```
typedef struct { // Generic Routine Activation Record
  void      (*step)(rar_t *);  // Pointer to step function
  uint16_t  pc;                // Saved control state
  rar_t     *caller;           // Caller's activation record
  uint16_t  children;          // Number of running children
  uint32_t  priority;          // Order in the ready queue
  uint8_t   depth;             // Index of LSB of our priority
  bool      scheduled;         // True when in the ready queue
} rar_t;

typedef struct { // Generic Variable
  void      (*update)(cv_t *); // Perform scheduled update
  trigger_t *triggers;         // Doubly-linked list
  uint64_t  last_updated;      // When variable was written
  uint64_t  event_time;        // Time of scheduled update
} cv_t;

typedef struct { // Int Variable
  // ...all the fields from cv_t...
  int value;        // Current value
  int event_value;  // Value to be assigned at event_time
} cv_int_t;

typedef struct { // Variable Trigger
  rar_t     *rar;       // Triggered routine
  trigger_t *next;      // Next trigger for this variable
  trigger_t **prev_ptr; // Back pointer for doubly-linked list
} trigger_t;
```

Fig. 6. Principal C data types used by the SSM runtime. Each routine has its own routine activation record type struct that begins with all the fields in an *rar_t* so the runtime can manipulate routine activation records generically. Similarly, each type of variable, such as the *int* shown here, has its own variable type struct that begins with the fields in *cv_t*.

```
examp(&a)
  var loc = 0
  wait a
  loc = 42
  after 10ms a = 43
  fork foo(42, loc)
  fork foo(40, loc) bar(42)
```

```
typedef struct {
  // ...all fields from rar_t...
  cv_int_t *a;       // Pass-by-ref
  cv_int_t loc;      // Local int
  trigger_t trig1;// Trigger
} rar_examp_t;

rar_examp_t *enter_examp(
 rar_t *caller, uint32_t priority, unit8_t depth, cv_int_t *a) {
  rar_examp_t *rar = (rar_examp_t *)
      enter(sizeof(rar_examp_t), step_examp,
                  caller, priority, depth);
  rar->a = a;             // Store pass-by-reference argument
  rar->trig1.rar = (rar_t *) rar; // Initialize our trigger
}
void step_examp(rar_t *gen_rar) {
  rar_examp_t *rar = (rar_examp_t *) gen_rar;
  switch (rar->pc) {
  case 0:
     initialize_int(&rar->loc, 0);                // var loc = 0
    sensitize((cv_t *) rar->a, &rar->trig1); // wait a
    rar->pc = 1; return;
  case 1:
    if (event_on((cv_t *) rar->a)) { // if @a then
      desensitize(&rar->trig1);       // De-register our trigger
    } else return;
    assign_int(&rar->loc, rar->priority, 42); // loc = 42
    later_int(rar->a, now+10000, 43); // after 10ms a = 43
    rar->pc = 2; // Single routine call: foo(42, loc)
    call((rar_t *) enter_foo((rar_t *) rar, rar->priority,
                                rar->depth, 42, &rar->loc));
    return;
  case 2:  // Concurrent call: fork foo(40, loc) bar(42)
    { uint8_t new_depth = rar->depth - 1; // 2 children
      uint32_t pinc = 1 << new_depth;
      uint32_t new_priority = rar->priority;
      fork((rar_t *) enter_foo((rar_t *) rar, new_priority,
                             new_depth, 40, &rar->loc));
      new_priority += pinc;
      fork((rar_t *) enter_bar((rar_t *) rar, new_priority,
                                new_depth, 42)); }
    rar->pc = 3; return;
  case 3:  ; }
  leave((rar_t *) rar, sizeof(rar_examp_t)); // Terminate
}
```

Fig. 7. Contrived *enter* and *step* functions illustrating the use of the API

IV. An SSM Runtime System

Our runtime system consists of a generic scheduler that dispatches C functions performing each routine's computation. Each routine is implemented as a pair of C functions: the *enter* function initializes the routine-specific activation record type, and the *step* function performs the work of the routine in a single instant of time, i.e., between when it is awakened by some event and when it suspends.

Our runtime system relies on two priority queues. The event queue stores (time, variable, value) triples ordered by increasing time. At the beginning of an instant, all the events at the current time update their variables and leave the queue.

The ready queue avoids unnecessary work for routines that do not need to resume. While SSM's formal semantics prescribe walking through the tree of running routines and checking the *wait* or *fork* statement at which each routine is suspended to decide which ones to run, the ready queue avoids this walk by maintaining a set of routines that will *definitely* run in the current instant, updated every time a variable is written. Routines waiting on a variable are held in the variable's list of *triggers*. Writing a variable adds each routine in its trigger list to the ready queue, which is prioritized according the order the routine would appear in the active routine tree traversal. We encode pathnames as left-justified 32-bit integers, ordered by their tree traversal order.

We represent time in μs with 64-bit integers. This avoids wraparound; 32-bit integers would wrap around in two hours.

Fig. 5 lists the interface SSM routines use to interact with the runtime; Fig. 6 shows the principal data types used to manage routines and variables; Fig. 7 illustrates how enter and step functions use the interface in a simple example.

Our GitHub repo: https://github.com/sedwards-lab/ssm

A. Routine Activation Records, Enter and Step Functions

To allow routines to freely suspend and resume, step functions store their state in an activation record struct, rather than using C's runtime stack. Each routine has its own specialized type, but all start with the *rar_t* fields listed in Fig. 6 so that the scheduler can treat them generically. Specifically, it is always safe to cast a pointer to routine-specific activation records to an *rar_t* pointer. Fields in *rar_t* hold a pointer to the routine's step function, its control state (an encoded program counter, which the step function uses to remember where to resume when it suspends), a pointer to its caller's activation record, how many of its children are currently running (used to determine when a *fork* has terminated), two numbers related to its scheduling priority (pathname), and, to avoid duplicates in the ready queue, a Boolean indicating whether this particular invocation of the routine is already scheduled.

Fig. 7 shows the activation record, *enter*, and *step* functions for the *examp* routine. The *rar_examp_t* struct begins with the fields in *rar_t*, but also includes a pointer to an integer variable *a*, a local integer variable *loc*, and a single trigger (the widest *wait* dictates the number: if a routine has a *wait* with 5 variables, it needs 5 triggers). The *enter_examp* function takes a pointer to its caller's activation record, two priority-related arguments we describe below, and a pointer to the *int* variable *a*—a pass-by-reference argument.

Fig. 5 lists the routine-management functions: *enter* allocates the activation record for a routine and initializes its generic fields; *call* immediately runs a particular routine (used, e.g., to call a routine as if it were a function); *fork* schedules a particular routine for execution in the current instant; and *leave* is the complementary function of *enter*: it deallocates the given routine activation record and, if it was the last running child of the caller, immediately transfers control back to its caller.

Only the *enter* and *leave* functions ever allocate or deallocate memory. Thus, provided a SSM system does not make an unbounded number of routine calls, it will not consume an unbounded amount of memory. At the moment, *enter* and *leave* simply use the C standard library's *malloc* and *free*; see Section VI for our thoughts about alternatives.

B. Variables and Triggers

A variable in SSM behaves like a traditional variable in an imperative language (i.e., it returns the value most recently written to it), augmented with the ability to schedule a future assignment, and for suspended routines to be reawakened by writes to the variable. Like activation records, the struct for each type of variable begins with a set of common fields (*cv_t* in Fig. 6), followed by type-specific fields for its current value.

Each variable may have at most one new value scheduled for it in the future—an event—stored in the *event_time* and *event_value* fields (*event_time* is *ULONG_MAX* when no event is pending). To enable the scheduler to update arbitrary variable types, each variable includes a pointer to a type-specialized *update* function that copies *event_value* to *value*.

Each variable maintains a list of triggers—routines awakened by a write—in the form of a doubly-linked list of *trigger_t* structs, rooted at a variable's *triggers* field. Each trigger is just a pointer to the activation record for the triggered routine. *Sensitize* and *desensitize* add and remove triggers.

Triggers are stored in a routine's activation record and can be reused across suspension points. Each trigger's *rar* field is initialized in its routine's *enter* function. Before a routine suspends, it calls *sensitize* on each variable it wants to watch. When a routine is awakened, it calls *desensitize* on all the triggers it sensitized. If a routine wants to wait for a particular value to be written to a variable, the routine will check for both an event and the desired value on a given variable and only desensitize its triggers if the desired condition holds. Otherwise, it will immediately suspend itself again without updating its control state or triggers. The code for *step_examp* in Fig. 7 illustrates *sensitize* and *desensitize*.

Type-specialized *assign* functions write the *value* and *last_updated* fields of a variable and schedules all the later routines currently sensitized to that variable. In addition to the value being assigned to the variable, *assign* also takes the priority of the current routine, and only schedules sensitive routines with a higher priority number to avoid looping behavior among routines in a single instant.

The *event_on* function compares the *last_updated* field to *now* (the current instant) to determine if a variable was written.

The type-specific *later* function schedules a future variable update by writing to a variable's *event_value* and *event_time* fields and adding the variable to the event queue. If *later* is called on a variable that already has a pending event, the existing event is overwritten. These semantics avoid an unbounded accumulation of events, but may surprise the programmer. We may have this raise a runtime warning error.

C. Single Routine Calls

Routines may recursively call other routines using a mechanism similar to that in C and related imperative languages. Fig. 7 illustrates the code at a single call site for a routine called *foo*. First, the "return address" is stored in the routine's activation record (*act−>pc*). Next, to allocate and initialize *foo*'s activation record, *enter_foo* is called with the pointer for the current activation record (saved in the callee's *caller* field), the current routine's *priority* and *depth* (when just a single routine is called, the callee simply inherits these values), and the actual arguments to *foo*: here, the literal 42 and a reference to the (passed-by-reference) *loc* variable.

Once *enter_foo* creates the activation record for *foo*, it is passed to *call*, which immediately sends control to the *step_foo* function (i.e., the body of the *foo* routine; not shown).

The *step_foo* function may suspend and be later awakened by a trigger, or may run to completion in the current instant. In either case, *step_foo* terminates by calling *leave*, which frees its activation record and returns control to its caller (whose activation record is in *foo*'s activation record's *caller* field).

D. Concurrent Routine Calls and Priorities

A routine may also invoke multiple concurrent routines. The mechanism is a generalization of single routine calls, and is also illustrated in Fig. 7. At a concurrent call site, each child's *enter* function is called, but the activation records these produce are passed to the *fork* function, which adds them to the ready queue according to their priority. The first child inherits the priority of the caller; the others are given successively higher priority numbers and thus will be executed in order.

The caller regains control after all children have called *leave*. This is managed by the *children* field in the caller's activation record, which is set to zero when a routine is first entered, incremented by *enter*, and decremented by *leave*, which immediately passes control back to the routine once all its children have terminated.

Deterministic concurrency was a key SSM design goal. We achieve it by assigning a unique priority numbers to each active routine and having the scheduler always run them in that order. In the case of a single routine call, the child simply inherits the priority of its parent, which is unambiguous because only one is ever running at a time.

When multiple routines are called, we assign priority numbers in a hierarchical manner that subdivides the range of priority numbers occupied by the caller. Each routine 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 routine calls k children, it assigns pairs (p,d'), $(p+2^{d'},d')$, $(p+2\cdot 2^{d'},d')$, …, $(p+(k-1)2^{d'},d')$, 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 routine 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 Fig. 7, the *new_depth*, *pinc*, and *new_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.

E. The Scheduler

The scheduler maintains two priority queues: the event queue, which holds variables scheduled to be updated, ordered according to their *event_time* fields; and the ready queue, which holds routine activation records scheduled to run in the current instant, ordered by increasing *priority* fields. We implement both as binary heaps whose maximum size can be determined from analyzing the program's dynamic call graph.

The *tick* function runs the system for an instant by performing all the variable updates queued for the instant then running every routine in the ready queue in priority order.

In the first phase, performing an event consists of removing the variable at the front of the event queue provided it is scheduled for the current instant *now*, calling its update function to update its *value* field, updating its *last_updated*, and then adding each routine waiting on the variable to the ready queue if it is not already there. This process stops when there are no pending events on the queue for the current time instant. Note that each variable's list of triggered routines is not modified during this phase: the routines themselves are exclusively responsible for managing their triggers.

In the second phase, the routine with the lowest priority number is removed from the ready queue and its step function invoked. The step function, in turn, may cause routines with equal or higher priority numbers to be added to the ready queue, either through a call to *schedule* (which may schedule another routine at the same priority) or through an *assign* call to a variable that triggers other routines at higher priorities.

The scheduler will terminate unless one of the activated routines refuses to suspend. Routines may contain 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 SSM.

F. Interfacing with the Real World

Our prototype simulates an SSM model like any discrete-event model: simulation time is advanced to that of the next event in the queue, *tick* is called, and the process repeats.

A SSM model interacts with a real-world environment through variables passed to its *main* function. Each output variable is given a concurrently running output routine that waits for an update then transmits the new value to hardware.

Each input event is received by interrupt routine that timestamps and enqueues it. Contention between *tick* and the interrupt routines over the event queue needs to be considered. It may be more efficient for the interrupt routines to place events in a separate incoming event queue that has fewer contention issues since it can be a simple FIFO.

After each call of *tick*, the system will determine the time of the earliest queued event, schedule a timer interrupt for then, then wait to be awakened by the next interrupt, which can be either an input event or the expiration of the timer.

V. Related Work

The Ptides [2], [7] distributed real-time model based on discrete event semantics [8] inspired much of SSM. Ptides is actor-oriented and reactive [9] where parallel processes (like SSM routines) communicate via timestamped events (like SSM variables). Ptides's sparse time model enables its PtidyOS [10] runtime to use an earliest deadline first scheduler. Like SSM's runtime, PtidyOS uses an event queue to spare the *tick* function from running in every instant. However, PtidyOS requires a fixed topology, precluding SSM's recursion.

A key advantage of SSM is its use of a separate semantics (i.e., one not based on events) for dealing with events in a single instant, unlike other discrete-event models. This gives SSM its determinism and sidesteps such infelicities as VHDL's delta cycles [6] and Ptides's microsteps and depth numbers.

SSM's labeling of instants with natural numbers and its intra-instant semantics were inspired by the synchronous languages [1]. This regular, tick-based approach works well for continuously evolving systems, but places an undue computational burden on reactive applications with sparse, irregular computation. SSM differs in that events are assumed to be sparse—in most instants, no computation takes place.

Hanxleden, Bourke, Girault, and others [3], [4] 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 [3] 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. SSM uses a much more efficient priority queue that avoids each delay having to do something at each tick.

We admire Esterel's deterministic, parallel semantics [11] but did not want its single-value-per-instant rule for signal values, which is difficult to explain and compile [12], causing it to reject programs with read-modify-write behavior (Fig. 4). Sequentially constructive concurrency [13], [14] attempts a fix by relaxing the single-value-per-instant rule, but is more complex. SSM uses a simpler syntactic total order.

Boussinot's ordering of concurrent routines to achieve determinacy, at the heart of his Reactive C [15], FairThreads [16], and FunLoft [17], inspired us. However, Boussinot's execution strategy searches for Esterel-inspired fixed-points, using a round-robin cooperative scheduler that repeatedly evaluates concurrent routines in order until they quiesce. For example, in SSM, *fork 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 its execution time is difficult to bound. Also, confusingly, the cyclical order of concurrent routines still matters in Boussinot's world. SSM adopts a more rigid, faster policy that is simpler to explain.

The graphical education language Scratch [18] provides delays, concurrency, and events, and is nearly deterministic. Its interpreter imposes cooperative multitasking among threads, which relinquish control at delays and at the end of looping constructs, making it possible (if a bit inefficient) to write "**for** (;;) **if** (a == 1) { ... }" in a cooperative multitasking setting without blocking the other threads. For efficiency reasons, we did not adopt such a forgiving execution semantics.

Verilog [19] and VHDL [6] 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). SSM's assignment and *after* parallel Verilog's blocking and nonblocking 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. SSM allows one event per variable; VHDL's ′event attribute inspired SSM's @ operator.

While designing SSM, we considered VHDL's *wait*, which can behave like SSM's *wait*, wait for a condition (e.g., *wait until CLK′event and CLK='1'*), or wait for a period of time (e.g., *wait for 10 ns*). The FreeHDL compiler [20] implements SSM's *wait*, transforms conditional waits into in a loop that checks the condition and suspends if it is not met, and transforms timeouts into waiting for a scheduled event (Fig. 3).

VI. Conclusions

We presented the sparse synchronous model via a toy imperative language with concurrent function calls and waits on variable updates, 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.

At the moment, our runtime system uses *malloc* and *free* calls to manage its tree-structured "cactus stack" of activation records, but we have two ideas on how to improve this. One approach employs fixed-sized stack pages to avoid fragmentation. Simple recursive calls would allocate their activation records on the same page and switch to a new page to avoid overflow; concurrent calls would allocate new pages. A segregated memory manager would be another fast approach that avoids fragmentation: pages are divided into fixed sized bins, and each activation record is held in the next largest bin that will fit it. Since these sizes are known at compile-time, the page and bin sizes can be specialized to each application.

One reviewer suggested we allow a program to check its progress by comparing logical time with real time. While this intriguing idea could enable systems to adapt to and make better use of their resources, it introduces a serious testing challenge by making behavior strongly dependent on details of platform execution time. Perhaps a Boolean "deadline missed" input could achieve similar results yet maintain testability.

Other attractive ideas include applying the distributed implementation techniques proposed for Ptides to SSM, and statically determining when we can relax SSM's strict child-ordering rules without introducing nondeterminism—perhaps using Rust-like ownership types—to enable true parallelism.

Our end goal is a user-friendly language built on SSM's semantics. We do not recommend coding in the toy language in Fig. 1; instead, we are currently developing a richer language.

Acknowledgments

We wish to thank Adam Kepecs, Reinhard von Hanxleden, and the FDL reviewers for their helpful suggestions.

References

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

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

[3] R. von Hanxleden, T. Bourke, and A. Girault, "Real-time ticks for synchronous programming," in *FDL*, Verona, Italy, Sep. 2017.

[4] T. Bourke and A. Sowmya, "Delays in Esterel," in *Proceedings of SYNCHRON*, Schloss Dagstuhl, Germany, Nov. 2009, Seminar 09481.

[5] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads Programming: A Posix Standard for Better Multiprocessing*. O'Reilly Media, 1996.

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

[7] J. Zou, "From Ptides to PtidyOS, designing distributed real-time embedded systems," Ph.D. dissertation, U. California, Berkeley, May 2011.

[8] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Annals of Software Engineering*, vol. 7, pp. 25–45, 1999.

[9] E. A. Lee, S. Neuendorffer, and M. J. Wirthlint, "Actor-oriented design of embedded hardware and software systems," *Journal of Circuits, Systems and Computers*, vol. 12, no. 3, pp. 231–260, Aug. 2002.

[10] J. Zou, S. Matic, and E. A. Lee, "PtidyOS: A lightweight microkernel for Ptides real-time systems," in *Proceedings of Real-Time Technology and Applications Symposium (RTAS)*, Apr. 2012, pp. 209–218.

[11] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[12] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer, Jan. 2007.

[13] R. von Hanxleden et al., "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation," *ACM Trans. Embedded Comp. Sys.*, vol. 13, Jul. 2014.

[14] S. Smyth, A. Schulz-Rosengarten, and R. von Hanxleden, "Practical causality handling for synchronous languages," in *Proc. Design, Automation, and Test in Europe (DATE)*, Florence, Italy, Mar. 2019.

[15] F. Boussinot, "Reactive C: An extension of C to program reactive systems," *Software: Practice and Experience*, vol. 21, no. 4, Apr. 1991.

[16] ——, "FairThreads: mixing cooperative and preemptive threads in C," *Concurrency and Computation: Prac. Exper.*, vol. 18, no. 5, Apr. 2006.

[17] ——, *Safe Reactive Programming: The FunLoft Language*. Lambert Academic Publishing, 2010.

[18] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transcations on Computing Education*, vol. 10, no. 4, p. 16, Nov. 2010.

[19] *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364–1995)*, IEEE Comp. Soc., 1996.

[20] E. Naroska, "The FreeHDL compiler/simulator system," Online at http://freehdl.seul.org/, Nov. 1998.