Concurrent ML as a Discrete Event Simulation Language

Abstract

Two common approaches to discrete event simulation, event scheduling (ES) and process interaction (PI) share a fundamental reliance on creating and dispatching continuations. In ES, the model builder implements individual steps of the simulation as procedures or methods, then explicitly schedules the steps by making calls to delay and queueing operations. Modeled entities’ behavior is spread across these procedures and ultimately depends on the way that the procedures are chained together by explicit enqueueing of future actions. Thus ES models tend to obscure the relationships between actions that an entity undertakes. ES typically has low time and space overheads associated with creating, enqueueing, dequeueing, and executing events.

In PI, the behavior of an entity over its lifetime is represented as a program thread. Interactions of simulated entities are modeled by interactions between threads using common concurrent programming mechanisms such as semaphores and mailboxes. The sequential behavior of modeled entities is directly reflected in the structure of the code that implements the model. The relatively large amount of memory needed to represent a thread in most systems, along with time required to switch between threads, gives PI simulation a reputation of requiring longer execution times and more memory than an ES simulation for a given model.

Concurrent ML (CML) uses Standard ML of New Jersey’s ability to capture lightweight continuations in order to provide a concurrent programming model in which threads require little memory and in which thread switching is fast. We demonstrate that CML can be easily adapted for use as a discrete event simulation language simply by equipping it with a mechanism for managing simulated time. In the resulting system, simulation code can make use of all of the process interaction mechanisms of CML itself.

1 Introduction

Three world views are prevalent in discrete event simulation implementations: event scheduling, process interaction, and activity scanning, [1, p. 74 ff.]. In both the event scheduling and process interaction techniques simulated time advances by leaping to the next point where there is simulated work to be done. By way of contrast, in activity scanning simulated time advances in one tick increments: after each advance the collection of activities is scanned to see which are enabled in the current state and time. The process interaction world view is the primary focus of this paper.

In the process interaction (PI) world view, sequential activity of modelled components is represented by sequential computational processes, [2]. When a modelled activity needs to wait, the process that represents it waits. Runtime systems for PI provide queueing, interaction and delay mechanisms that control the advance of simulated time.

In contrast, the event scheduling approach fundamentally exposes to the model builder a time-ordered future event list (FEL) consisting of code chunks to be executed when their due-time is reached. Similarly, the model builder explicitly can arrange for code chunks to be placed on wait lists and can release them when some condition is satisfied in the simulation. Unfortunately, models implemented in ES-style suffer from what Reppy has called, in the context of interactive programs, “inverted structure”: control flow that is conceptually sequential has to be broken up by the model implementor into small chunks that are explicitly linked to one another. The chunks are then woven into an execution schedule by a behind-the-scenes event loop, [3, p. 145]. One consequence of this approach is that it becomes difficult to modify simulation models because modeled behavior is spread across many code segments. It is also difficult to incorporate code from real systems into models. Nevertheless, event scheduling (ES) is often favored for its relative simplicity of implementation and its performance.

Internally, PI simulation uses future event lists and wait lists, but instead of modeler-written code chunks, the lists contain representations of the suspended processes. Process
interaction simulation is often implemented using *threads* provided by the implementation language’s run-time system. For simulators implemented in Java, for example, Java’s threads are used. In these systems the amount of storage required for representing a thread and the amount of time required to switch between threads is large compared to the way ES represents events and switches between them. As a result, PI has a reputation as being more costly in time and storage than ES for a given simulation model, [1, p. 524].

A second difficulty with PI systems is that even though they are based on the threads supplied by the implementation language, the process interaction mechanisms available to the model builder typically are *not* the thread interaction mechanisms of the implementation language. Thus J-SIM, [4], JavaSim, [5], JiST, [6], and CSIM for Java, [7], do not provide for use of Java synchronized blocks and conditions in simulations. Instead, each simulation package provides its own mechanisms for process interactions. It might be argued that the simulation systems’ mechanisms are superior to Java’s in some way. However, lack of support for the underlying language’s mechanisms makes it difficult to move code into the simulated system from real systems. It also makes the lore that develops concerning concurrent programming techniques in a particular language inapplicable in the simulation system. The failure to use a host language’s standard process interaction semantics is undoubtedly because the implementations of the synchronization mechanisms are not considered user-programmable components of the language, although in JiST ambitious rewriting of Java class files is done to implement the interaction semantics used in that system.

The choice to explore simulation in Concurrent ML stems from two observations. First, the essence of both ES and PI simulation is *constructing and dispatching continuations*. In ES, the continuations are explicitly constructed by the modeler and have light-weight representations. In PI, the continuations, represented by thread state, are automatically constructed by the system when needed and represented in relatively heavy-weight threads. The similarity of the underlying mechanisms suggests that a language with threads based on light-weight continuations might be ideal for building high performance, large scale PI simulations. Second, the notion of time (and timeouts) can be treated orthogonally to other aspects of interactions between processes. Any process interaction can optionally have a timeout attached to it.\(^1\)

A language that treated the notions of time and timeouts orthogonally to other aspects of process interaction might allow a PI simulation system supporting the host language’s process interaction model to be easily constructed. Concurrent ML (CML), [3], satisfies both of these criteria. CML is a concurrency library for the SML of New Jersey (SMLNJ), [8], implementation of the Standard ML language (SML). [9] CML uses SMLNJ’s lightweight continuations to implement threads. CML’s process interaction mechanisms treat all thread interactions orthogonally, including timeouts. An introduction to the interaction model is given below. Because CML’s threads are so light-weight and the interaction model so convenient, it has previously been observed that CML encourages software system designs where each naturally sequential activity in the problem space is mapped to a separate thread in the program, [10].

Based on these observations we have modified CML to support simulation. The modifications consist of: providing a notion of simulated time; adjusting the process scheduling algorithms to achieve reproducibility in simulations; and labelling processes as either simulation processes or normal processes allowing mixed executions in which some processes implement the simulation (in simulation time) and others observe the simulation and report on it (in wall-clock time). The latter is an example of embedded simulation, [11].

The next section introduces SML and CML, after which we describe the adaptations made internally to CML to support simulation and the resulting interface for constructing simulation models. Section 5 presents a simple simulation model constructed in the system and illustrates how a model can interact with non-simulation threads. Section 6 presents preliminary performance numbers supporting the viability of the approach and section 7 suggests directions for further exploration.

## 2 Standard ML

SML is a general purpose programming language originally developed for use in exploring logics related to programming. It is generally described as being a strongly typed, polymorphic, and mostly functional language. It has an extensive module system with polymorphic features that substantially reduce the amount of nearly-duplicate source code that frequently arises in large systems. SMLNJ is one of several extant implementations of SML. SMLNJ compiles SML to native code for several different processor families. Its run-time system (written in C) provides garbage collection and access to host operating system services. SMLNJ augments SML with a call-with-current-continuation operator (callcc) and a throw operator to transfer control to a previously captured continuation, [12].

---

\(^1\)Time and timeouts in Java are not orthogonal to the process interaction mechanisms because entrance to synchronized blocks is not subject to a timeout but wait operations can have timeouts associated with them.
The CML library uses callcc and throw to implement threads. Since the run-time system does not use pre-allocated, fixed-size stacks as is typical of C and Java thread implementations, CML threads can use very little storage, yet any particular thread has available to it as much storage as it needs up to the host operating system limit.

CML couples its lightweight threads with an expressive collection of thread interaction mechanisms such as synchronous channels, mailboxes, synchronizing shared memory (syncvars), [13], and of course, the passage of time. All of the mechanisms are built around a common notion of first-class synchronization event values and event combinators. Synchronization events represent the potential for a synchronization interaction, so an interaction can be described separately from an instruction to perform it. Each CML interaction mechanism is characterized by the behavior of the events it supports. The mechanisms include:

- synchronous channels with unbuffered send and receive events. A send event on a channel must be matched with a corresponding receive event for either to occur.
- the iVar variant of syncvars with put and get events with write-once semantics. Getting from an empty iVar will block; writing to a previously-written iVar is an error.
- the mVar variant of syncvars with put, get and take events. As for iVars an attempt to get (or take) from an empty mVar will block. However a successful take returns an mVar to the empty state whereupon another put can be done. There is also an atomic swap operation on mVars.
- mailboxes with buffered send and receive events. Multiple send events may be performed on a mailbox without any receive events being performed.

Separately from these interaction mechanisms, time is embodied in timeOut and atTime events, the former representing the passage of a given amount of time and the latter the attainment of a specified global time. The CML library also provides event constructors for interacting with operating-system-based resources such as sockets. The entire CML mechanism leverages ML polymorphism. Events, channels, syncvars, mailboxes, RPC protocols, etc. are all polymorphic in the type of the value that is conveyed in the interaction.

Any individual event can be performed by using the sync operation on it. Performing an event involves perhaps blocking the thread for awhile followed by delivery of the event’s result value to the thread in the form of the result of the call to sync.

Event combinators produce new event-type values from other events. For example, wrap(evt,f) is an event that has the synchronization behavior of evt followed by the evaluation of function f on the result of the event. The result of sync(wrap(evt,f)) is the result returned by the function f evaluated on the result of evt.

The choose combinator gives CML much of its expressive power concerning process interactions. It produces a new event from a list of events; the new event represents potential synchronization with any one of the listed events. The sync operation applied to a choose event performs exactly one of the events given to choose. The fact that time shows up in CML only as the timeOut event and atTime event is the basis for the claim that time is orthogonal to other process interaction mechanisms in CML: time-related events can be combined with any other events using choose to put a timeout on any thread interaction.2

Use of CML is initiated from non-threaded SML by calling RunCML.doIt(f, NONE). This sets up the run-time support for threads and invokes function f in the initial thread. The NONE parameter indicates that we accept the system default time quantum.

3 Adapting CML to Support Discrete Event Simulation

It is worth repeating here: the goal of this work was not merely to support process interaction simulation in CML. Rather, it was to support process interaction simulation in which the simulation processes have the full power of the CML thread interaction mechanisms available to them. This actually turned out to be a simplifying requirement: there was no temptation to implement home-grown interaction mechanisms. However, this goal also required us to go below the CML interfaces which (wisely) do not expose enough of CML’s inner working to allow implementation of the internal protocol by which CML coordinates decisions regarding blocking and choosing a single event from a choose list.

Once the necessity of working inside CML is accepted a little reflection shows that the principle issue is “how is simulated time to be advanced?” Recalling that simulated time

\[2\] In the examples below we use CML-provided abbreviations to avoid writing sync explicitly. For example, an iVar is a synchronizing shared memory value with events that allow it to be written once (iPutEvt) and read multiple times (iGetEvt). Performing an iGetEvt blocks the calling thread if the iVar has not yet been written. If v is an iVar, (iGet v) means the same thing as (sync (iGetEvt v)). Similarly, (select L) is the same as (sync (choose L)).
advances when there is no work to do at the current simulated time, the question becomes “what mechanisms will support

1. determining that there is no work to do at the current simulated time and

2. determining the next time at which work can be done?”

The first part of the question is answered by keeping track of ready simulation threads and ready non-simulation threads separately. A field was added to thread descriptors to allow each thread to be tagged as a simulation thread or a non-simulation thread. If during execution there are no ready simulation threads then there is no more work to be done at the current simulated time and the time should be advanced.

The second part of the question is answered with a future event list ordered by time of occurrence. Note that since time advances only when there are no ready threads at the current simulated time the only possible place to find a thread to execute is the FEL. Entries are created on the FEL when threads attempt to synchronize on an event that involves the passing of time–be it a basic timeout event or an event constructed by a combinator. Simulate-time events follow CML’s existing approach to time-related events: we simply added support for timeOut and atTime events in simulated time instead of wall-clock time. The implementation of CML’s internal synchronization protocol was adopted directly from the existing timeOut and atTime events. There was, however, a performance problem that had to be addressed for the queue itself. In ordinary CML applications there are usually only a few threads with outstanding delay events. Most threads are either on the ready list or waiting for something other than the passage of time so the timeout list remains short and a simple ordered linked list suffices. For simulation however, the common case is that almost all the threads (potentially thousands) are on the FEL. The linked list is much too slow for this situation. Therefore, a heap-based priority queue is used for the simulated-time FEL.

The CML scheduler calls the SimTimeOut module to poll the FEL when there are no ready simulation threads. When the FEL is polled it advances simulation time to that of the next event and returns the event. The scheduler dispatches the continuation carried in the returned event.

Finally, we wished to provide reproducible simulations, so one further change was needed in the scheduler. The CML scheduler is a pre-emptive, time-slicing scheduler. When a thread’s time-slice expires it is placed at the end of its priority queue, implementing a round-robin within priority scheme. Since the (wall-clock) timer interrupts are not synchronized to simulation execution this wreaks havoc on reproducibility. To prevent this undesirable behavior the scheduler places interrupted simulation threads at the head of the simulation ready queue. With this change, behavior within the class of simulation threads is as if there were no preemption, i.e., deterministically, but preemption still occurs and allows non-simulation threads to run as they become ready.

In the final design, then, simulation events strictly in the future are maintained on a time-ordered queue in SimTime-Out. Events to be executed at the current time are maintained in the thread scheduler’s priority 0 ready queue. Events involving process interactions are on queues associated with interprocess communication mechanisms, e.g. channels, syncvars, or mailboxes, themselves. In all cases the code associated with an event is represented by a continuation that is constructed in the normal CML way by the run-time system when user threads invoke CML primitives.

4 Simulation interface

The simulation-related functions are packaged in the CMLSim module (called a structure in ML) for use in constructing simulation models. See Fig. 1. Simulation threads can use all of the capabilities of CML except wall-clock delays. Since a wall-clock delay makes the thread not ready, simulation time will advance unpredictably in a thread that invokes a wall-clock delay. The CMLSim.spawn* functions encapsulate the use of priority 0 to represent simulation threads. It is merely an implementation detail that need not concern the modeler. Functions timeOutEvt and atTimeEvt create a new event occurring at a future simulated time. As is usual in CML these events represent the possibility of delay rather than an actual delay. An actual delay occurs only if the event is later synchronized (as in the delay function which is included in CMLSim as a convenience since explicit delay operations are common in simulation models). Finally, CMLSim.doit is provided as a convenient replacement for RunCML.doit (the usual way of starting a CML program), handling the detail of ensuring
structure CMLSim = struct
(* Priority 0 threads are
simulation threads *)
val spawn = CML.spawn_prio 0
fun spawnc f a = CML.spawnc_prio 0 f a
val timeOutEvt = SimTimeOut.timeOutEvt
val atTimeEvt = SimTimeOut.atTimeEvt
fun delay t =
  CML.sync(SimTimeOut.timeOutEvt t)
(* It is essential that the main thread
be marked as being a simulation thread. *)
fun doit f = RunCML.doit
  (fn () => ignore (spawn f), NONE)
end

structure Customer = struct
    fun customer (f1,f2,r,doneCh) =
        let
        fun behavior () =
            let
            (* capture the arrival time *)
            val arrival = Report.arrival(r)
            in
            Facility.requestService f1;
            Facility.requestService f2;
            Report.departure (r, arrival);
            CML.send (doneCh, ()
            end
        end
        in
        CMLSim.spawn behavior
        end
end

Figure 1. The Simulation Interface

that the main thread of the simulation is tagged as a simulation
thread.

5 Example

This sections presents an example of a queueing system
model in which customers sequentially use a pair of single-
server facilities. The example is derived from the CSIM
example in [14, p. 122] and the SSF example in [1, p. 121].
Complete code for the customers is in Fig. 2. Each cus-
tomer measures and reports its own time in the system using
the Report structure (not shown). Note how the customer
function spawns a thread that embodies the customer’s be-
havior then immediately returns.

The facilities in this model are single servers preceded by
FIFO queues. The server and the queue, together, are mod-
eled in the Facility structure (Fig. 3). The queue is mod-
eled using a CML mailbox (recall that a mailbox provides
buffered communication). Each facility’s server is a thread
that takes a request from the mailbox, delays for a random
time, then posts the request as complete. Note how the deci-
sion to use a mailbox as a request queueing mechanism and
a SyncVar.ivar as the completion-notification mechanism
is abstracted away in the signature and encapsulated in the
requestService function. This is an example of a sim-
ple RPC protocol, [3, p. 94], but this detail is not apparent
outside of Facility. Models may require more instrumentation
of the queue than CML’s mailbox provides. Instrumented queue implementations are easily constructed at the
application level using the CML thread interaction model.
Such queues could be part of a simulation library or built
as-needed for a particular simulation model.

structure Customer = struct
    fun customer (f1,f2,r,doneCh) =
        let
            fun behavior () =
                let
                    (* capture the arrival time *)
                    val arrival = Report.arrival(r)
                    in
                        Facility.requestService f1;
                        Facility.requestService f2;
                        Report.departure (r, arrival);
                        CML.send (doneCh, ()
                    end
                end
        in
            CMLSim.spawn behavior
        end
end

Figure 2. Simulated customers

Given the Customer and Facility structures it is a rela-
tively simple matter to construct the overall simulation (Fig.
4). After creating the facilities the main thread spawns a
collector thread to collect completion messages sent over
a CML channel from the customers. When the collector
determines that the simulation is finished it sets the iVar
allDone. The main thread goes on to create the cus-
tomers using the function generate, waiting for a random
arrival time before creating each one. Finally it waits for
the allDone message from the collector thread to trigger its
final reporting.

The expression of customer behavior in Fig. 2 exemplifies
the relative clarity of PI models. Compare it to Fig. 5 which
illustrates how customers’ behavior might be expressed in
an event scheduling model. The simple, sequential cus-
tomer behavior is quite thoroughly obscured in the ES code.
The reader may wish to pause and consider the relative con-
sequences, for the PI and ES code, of changing the model to
use a third facility, or more interestingly, conditionally use
a third facility.

One use of mixed simulated-time and actual-time threads
is to easily animate or interactively control a simulation.
The eXene windowing toolkit, [15], from CML provides
a thread-based interface for the X-Window system that can
be used to build control and monitoring user interfaces for
simulations. For example, adding a progress meter to the
simulation of Fig. 4 involves the code of Fig. 6. The
queryLoop runs in a separate thread. Once per second it
accepts a message on the queryCh. The values in the mes-
sage are used to update a percent-done slider widget. In the
simulation itself, the collect function of Fig. 6 replaces
structure Facility: sig
  type facility
  val facility: (string*unit->Time.time)-> facility
  val requestService: facility->unit
end

= struct
  type facility = 
    {id: string, 
      fifo: 
        unit SyncVar.ivar Mailbox.mbox}
  (* serviceTime is a function delivering 
     the (random) service time for each 
     use of the facility *)
  fun facility (id,serviceTime) = 
    let
      val fifo = Mailbox.mailbox()
      fun loop () = let
        val doneVar = Mailbox.recv fifo
        val serviceTime = serviceTime()
        in
          CMLSim.delay(serviceTime);
          SyncVar.iPut (doneVar, ());
          loop()
        end
      in
        CMLSim.spawn loop;
        {id=id, fifo=fifo}
      end
    in
      requestService (f as {id, fifo}) = 
      let
        val doneVar = Mailbox.recv fifo
        val serviceTime = serviceTime()
        in
          CMLSim.delay(serviceTime);
          SyncVar.iPut (doneVar, ());
          loop()
        end
    end
end

Figure 3. Simulated facilities

Figure 4. Composing the Simulation

structure Sim = struct
  fun doSim (nCust, servTime, arrivTime, seed) = 
    let
      fun main() = let
        val rand1 = Random.rand(seed, 1037)
        fun serviceTime1() = Time.fromReal (servTime*(Random.randReal rand1))
        fun serviceTime2() = Time.fromReal (servTime*0.5*(Random.randReal rand1))
        fun arrivalTime() = Time.fromReal (arrivTime*(Random.randReal rand1))
        val r = Report.stats (*Single queue simulation*)
        val f1 = Facility.facility (*"F1", serviceTime1*)
        val f2 = Facility.facility (*"F2", serviceTime2*)
        (* doneCh is used by customers to 
          report their completion *)
        val doneCh = CML.channel()
        (* allDone is set when the collector 
          has collected all the customers *)
        val allDone = SyncVar.ivar()
        (* collect the customers. 
          Report when done *)
        fun collect () = let
          fun loop 0 = SyncVar.iPut (allDone, ())
            | loop n = 
              (CML.recv doneCh;
               loop (n-1))
          in
            loop nCust
          end (* collect *)
        (* generate the customers *)
        fun generate 0 = ()
          | generate n = let
            val interarrival = arrivalTime()
            in
              CMLSim.spawn collect;
              Customer.customer(f1,f2,r,doneCh);
              CMLSim.delay (interarrival);
              generate (n-1)
            end (* generate *)
          in
            (* body of main thread *)
            (* start the collector thread *)
            CMLSim.spawn collect;
            (* generate customers 
              using this main thread *)
            generate nCust;
            (* wait for them all to finish *)
            SyncVar.iGet allDone;
            (* report the statistics *)
            Report.reportStats r;
            Report.reportWallTime r
          end (* main *)
        in
          CMLSim.doit main
        end (* doSim *)
    end (* Sim *)
structure CustomerES = struct
  fun customer (f1, f2, r, finished) = let
    val arrival = Report.arrival(r)
    fun behaviorPart3 () = (Report.departure(r, arrival); finished())
    in
      (* This is the first part of the behavior *)
      FacilityES.requestService
        (f1, behaviorPart2)
      end (* customer *)
  end (* CustomerES *)

Figure 5. ES Customer Simulation

that of Fig. 4. The principle change is the select in the loop body to allow either receipt of a completion message from a customer or to send a status message on the query channel. The other changes to the collector loop merely allow it to continue providing messages to the query channel after all of the customers have finished.\(^6\)

6 Performance

The principle emphases in this work have been ensuring that process interaction models are easily expressed, that they can use all of the CML process interaction mechanisms orthogonally, that simulation executions are reproducible, and that large numbers of interacting simulated entities can be supported. Preliminary performance testing shows that on a 2.4GHz Pentium the example simulation above takes less than 20 microseconds of wall-clock time per simulated customer. Each customer, represented by its own thread, is enqueued (in the mailboxes) twice, suffers a time delay in each of the two facilities, and finally reports its completion to the collector thread. Internally the data structures used for queueing ready threads and the mailboxes are efficient, amortized constant-time-insertion-and-removal queues. The system supports one hundred thousand simulated entities (threads) in less than 500MB of virtual address space.

\(^6\) Note that the unary negation operator in SML is \(~\).
7 Conclusion and future work

With a small investment in programming effort CML has been adapted for discrete event simulation. The resulting system supports process-interaction style simulation with low per-entity cost, yet allows simulations to use all of the process interaction features of the CML system. Simulated-time threads and actual-time threads can interact to support embedded simulation as well as dynamic monitoring and control of simulation progress. A patch file giving the most recent SMLNJ release discrete event simulation capability is available from the author’s website.

CML-based process-interaction simulation as presented here is, in its execution, very close to that of classical event-scheduling simulation: data objects are allocated and enqueued either on the scheduler ready queue or on a wait list. The objects being enqueued here are CML threads represented by continuations. In comparison with the size and creation cost of a Java thread or pthreads thread, a CML thread is lightweight indeed. Based on this observation one expects that CML-based simulations would have execution costs—time and memory—comparable to those of event-scheduling simulation. Experimentation is needed to more closely establish where CML-based simulation falls in the spectrum of simulator performance.

To make CML a more convenient simulation platform a utility library of simulation-related code would be desirable. For instance, in the example above the service and arrival times are taken from a uniform distribution because the existing random number libraries in SMLNJ don’t provide the exponential and normal distributions more commonly used in simulations. Also, there are likely to be common simulation tasks that should be captured in library code for re-use: facilities, simulation termination, and so forth. More ambitiously, network simulators such as NS2, [16], and SSFNet, [17, 18], provide extensive libraries supporting simulation of widely-used network protocols along with packages that produce network traces interpretable by tools like tcpdump.

The ability for simulated-time and actual-time threads to interact is attractive but it carries a caveat: if a simulation thread waits for a non-simulation thread then simulation time can advance unpredictably. The technique that we have used thus far relies on the programmer ensuring that interactions with non-simulation threads are always part of a choice list so that simulation time advances remain under the control of simulation threads. It would be useful to figure out ways, perhaps exploiting the type system, to prevent unintended interactions between the two worlds.

8 Acknowledgements

I am grateful to (blinded for review) who originally suggested that replacing SML’s Time structure with a simulated-time structure was a good first step toward creating a simulation infrastructure using CML. This work was supported in part by NSF grants (blinded for review) and (blinded for review).

References


