Programming with Managed Time

Sean McDirmid
Microsoft Research
Beijing China
smcdirm@microsoft.com

Jonathan Edwards
MIT CSAIL
Cambridge, MA USA
edwards@csail.mit.edu

Abstract

Most languages expose the computer's ability to globally read and write memory at any time. Programmers must then choreograph control flow so all reads and writes occur in correct relative orders, which can be difficult particularly when dealing with initialization, reactivity, and concurrency. Just as many languages now **manage memory** to unburden us from properly freeing memory, they should also **manage time** to automatically order memory accesses for us in the interests of comprehensibility, correctness, and simplicity. Time management is a general language feature with a large design space that is largely unexplored; we offer this perspective to relate prior work and guide future research.

We introduce *Glitch* as a form of managed time that **replays** code for an appearance of simultaneous memory updates, avoiding the need for manual order. The key to such replay reaching consistent program states is an ability to reorder and rollback updates as needed, restricting the imperative model while retaining the basic concepts of memory access and control flow. This approach can also handle code to enable *live programming* that incrementally revises program executions in an IDE under arbitrary code changes.

1. Introduction

Computers present us with a bleak experience of time: any memory location can be written at any time, determined only by global control flow built from jumps and branches. Programmers must then tediously coordinate state updates by carefully initializing objects so their fields are not read too early, reactively repairing views so that they are consistent with changing models; organizing analyses (e.g. compilation) into multiple passes so updates (e.g. symbol table entries) are visible where needed; locking resources that can be manipulated concurrently; and so on. Although immutability is often touted as a solution to these complexities, the ability to update state directly remains convenient and popular.

We propose that programming languages address state update order by abstracting away from the computer's model of time; i.e. they should **manage time**. We draw an analogy with **managed memory**: we now widely accept that languages should unburden us from the hard problem of correctly freeing memory. Languages should likewise correctly order state updates for us by hiding computer time.

Managed time as a concept relates and contrasts past and ongoing efforts to provide a useful perspective to assess an entire design space of possible solutions and suggest future research. Time is also a fundamental problem for improving the programming experience via *live programming* that provides programmers with continuous feedback about how code executes [23]. Managed time enables a "steady frame" view of program execution to reason about this feedback. It also aides in incrementally revising execution so live programming can be realized beyond select tame examples.

This paper presents Glitch, a new form of managed time where state updates appear to occur simultaneously through code replay. Glitch emphasizes programmability by retaining the familiar imperative programming model, restricting it instead with state updates that are both commutative (order free) and can be rolled back. Despite these restrictions, complex useful programs from compilers to games are still expressible. Additionally, Glitch is fully live: it enables live programming with time travel to past program executions, which are incrementally repaired in response to code edits.

We introduce Glitch by example in Section 2. Section 3 describes the techniques needed to realize Glitch while Section 4 presents our experience with it. Section 5 discusses the past and present of managed time; in particular, approaches such as Functional Reactive Programming (FRP) [16], Concurrent Revisions [5], Bloom [1], and LVars [29] tame state update with various techniques (reactive dataflow signals, isolation, strict monotonicity), making different tradeoffs. Section 6 concludes with a future of managed time.

```
trait Vertex:
  set _reach, _edges
  for v in _edges:
    _reach.insert(v)
    for w in v._reach:
      _reach.add(w)
  def makeEdge(v):
    _edges.insert(v)
```

Figure 1: The Vertex trait.

2. Glitch by Example

Consider three columns of code written in YinYang [32], a language based on Glitch with a Python-like [46] syntax:

```
cell x, y, z      y = 20      on e:
x = y + z         z = 22        z = 10
```

All reads and writes to variable-like cells, as well other state constructs, execute **simultaneously**, meaning all reads can see all writes. Before event e fires, reading the x cell anywhere will result in 42 even though x's assignment lexically precedes those of y and z. Computations that depend on the same changing state are updated at the same time; e.g. z becomes 10 when e fires, causing x to change from 42 to 30.

State definition and update in Glitch can be encapsulated in procedures and objects; consider:

```
trait Temperature:
  cell _value
  def fahrenheit: return _value
  def celsius: return 5 * (_value - 32) / 9
  def setFahrenheit(v): _value = v
  def setCelsius(v): _value = 9 * (v + 32) / 5
```

Here the Temperature trait (a mixin-like class) maintains a temperature that can be read or set in either Fahrenheit or Celsius units. Reads and writes to a temperature object's state, consisting of a private _value[1] cell, are simultaneous as in the last example; consider:

```
cell x, y                    y = 50
object tem is Temperature    tem.setFahrenheit(y)
x = tem.celsius
```

This code's execution assigns 10 to x. If y later changes from 50 to 70, x would become 21.11 at the same time.

State and updates can more safely be encapsulated in modules since simultaneous execution eliminates side effect ordering problems. Consider the definition of a Vertex trait in Figure 1. The statements in a trait execute during the construction of any object that extends the trait. Object construction executes simultaneously with the rest of the code, not only eliminating initialization orders, but also allowing them to encode object invariants; e.g. a Vertex object's constructor can observe and react to all state updates performed by makeEdge method calls on it.

Sets, like cells, are state constructs for expressing collections. A Vertex object's _reach set always contains all vertices in its vertex edge transitive closure. Computing a transitive closure in the presences of cycles normally requires explicit multiple passes because result sets must be read and written multiple times, but these passes are hidden from the programmer in Figure 1. Glitch instead replays the code until an iterative fixed point is reached; consider:

```
object a, b, c are Vertex      b.makeEdge(c)
b.makeEdge(a)                  c.makeEdge(b)
```

Glitch replays the Vertex constructor on object b at least twice so that its _reach set contains all a, c, and itself. Glitch automatically computes fix points for cyclic dependencies, which are common and often unavoidable. Much of Glitch's technical complexity goes into handling cyclic dependencies work negation and retracted state updates (see Section 3).

Simultaneous execution prevents the direct encoding of cyclic updates like `[x = x + 1]` that diverge since their own updates cause them to replay! Glitch instead provides accumulators as state constructs (like cells and sets) that support commutative updates (e.g. sums); consider:

```
sum x = 0      y = x      x += 1
cell y         x += 2
```

All += ops on a sum accumulator can be executed in any order. Intermediate x values are not observable so the y cell is assigned to 3 rather than 0, 1 or 2. Removing any += op will trigger a -= op to incrementally "rollback" the update. Section 3 discusses how both commutativity and rollback play big roles in all state constructs, not just accumulators.

Tick Tock Goes the Clock

Although Glitch's simultaneous execution hides computer time, programmers must still deal with "real" time as defined by discrete event occurrences; e.g. when a key is pressed, a timer expires, or a client request is received. Event-handling code executes differently from non-handler code: it only sees the state immediately prior to the event, and its updates are only visible after the event. With these semantics, a handler can encode a stepped state update like `[x = x + y]` that would normally cause divergence. However this update is **discrete**: only x and y's values at the event are read, while x's new value is only seen after the event; the update cannot see itself.

Consider Figure 2's Particle trait that extends objects with simple Verlet integrators using two last positions (position and _last) to compute a _next position with an implied velocity. Particle constraints are expressed as to positions in relax method calls that are interpolated from the implicit next position; the interpolated positions are then accumulated in the _relaxed sum accumulator and counted by the _count so that particles can be constrained by multiple relax calls.

Event handlers are encoded using the on statement. Particle handles global Tick events by assigning the particle's position to an average of its _relaxed positions. The particle's position at the event must also be saved in the _last cell so that it can be used in successive _next computations. The Particle trait can then be used as follows:

[1] Python's underscore practice is used to specify private members.

```
trait Particle:
  cell position = Vec(0, 0), _last = position
  sum _relaxed = Vec(0, 0), _count = 0
  def _next: return 2 * position - _last
  def relax(to, strength):
    _relaxed += _next * (1 - strength) + to * strength
    _count  += 1
  on Tick:
    if _count == 0: position = _next
    else: position = _relaxed / _count
    _last = position   # comment: save position for use in next step
```

Figure 2: The Particle trait.

```
object a, b are Particle      a.relax(Vec(5, 5), 0.025)
b.relax(a.position, 0.01)     b.relax(Vec(0, 0), 0.025)
```

This code creates two particles a and b, where a moves to (5,5) while b both chases a and moves to (0,0). On every Tick event, position and _last fields are discretely updated, creating a new result for _next that induces Relax call replays, adjusting the _relaxed values used on the next Tick event.

Many continuous behaviors must execute after an event occurs; e.g. consider opening a window that exists until dismissed. Event handlers can specify code that is subject to simultaneous execution in time following an event; consider:

```
on widget.mouseDown:
  cell pw = widget.position
  cell pm = MousePosition
  after:
    widget.position = pw + (MousePosition - pm)
    on widget.mouseUp:
      widget.position = widget.position # hold widget position
      break                             # stop the most inner after block
```

This code implements a typical UI widget mouse drag adapter that uses resolution resilient non-accumulated mouse position deltas. When the mouse button is pushed down on widget, the cell positions of the widget and mouse are stored in fresh cells. A continuous future-influenced execution is then specified as an after block that moves the widget with the mouse. Finally, when the mouse goes up, a break statement makes the after block stop. However, break causes all after block behavior to be rolled back, including the widget position assignment! To avoid reverting widget's position, its position is assigned to itself in the handler, holding the position via Glitch's discrete update semantics.

3. Making a Time Machine

Glitch's simultaneous execution is a powerful illusion conjured by tracing state dependencies as statements execute, and *replaying* them as needed so that all statements are guaranteed to see each other's state updates. A *task* is Glitch's incremental unit of replay; a complete program execution consists of a tree of tasks whose boundaries are explicitly specified in code. Pseudocode for task replay is shown in Figure 3. When a task is replayed (Replay), all of its existing

```
trait Task:
  def Replay():
    for u in updates:
      u.stale = true          # mark all existing updates as stale
    Exec()                    # do custom behavior of task
    for u in updates:
      if u.stale:
        u.Rollback()          # Roll back updates that are still stale
        for t in u.state.depends:
          t.Damage()
        updates.Remove(u)
  def Update(u):              # do state update during Exec
    if u.stale:
      u.stale = false         # update is existing
    else:                     # update is fresh
      if u.Install():         # perform the update
        updates.Add(u)
        for t in u.state.depends:
          t.Damage()
      else: ...               # does not commute, error!
  def Read(state):            # read state during Exec
    state.depends.Add(this)   # add as dependency to state
    return state.value
```

Figure 3: Imperative pseudocode for task replay; note this is standard imperative code rather than YinYang code.

updates are marked as *stale*. During task execution (Exec), updates are made through calls to Update, which either unstales the update as stale if it pre-exists, or otherwise installs it. If an update is still stale after Exec finishes, the update is no longer performed and so is *rolled back*. Rollback is update specific: an accumulator operation is reversed; a cell assignment is undone, and an inserted element is removed from a set. During Exec, a task is added as a dependency to any state it reads (state.depends) so it can be *damaged* when an update on the state is installed or rolled back. Glitch then repairs damaged tasks by replaying them, which can cause damage to more tasks, until a *fix point* is reached where no tasks need to be repaired. As an example, consider:

```
set s                  object x, y, z
task: # task:i         task: # task:j      task: # task:k
  if !s.contains(x):     s.insert(x)         if s.contains(y):
    s.insert(y)                                s.insert(z)
```

Tasks are defined in task blocks (here they are also named in comments). A possible execution of this code is shown in Figure 4. Assuming task:i (first-column) is played first, set s will not contains x and so y will be added to it. Next, task:j (second-column) plays, adding x to s to damage task:i, while task:k (third-column) plays adding z to s because s contains x. When task:i is replayed, s now contains x and s.insert(y) does not execute, becoming stale to be rolled back as per Figure 3's pseudocode. This rollback causes task:k to be damaged, whose replay then causes the addition of z to be removed from s. A more optimal replay order would be to play task:j first, avoiding replaying task:i and task:k more than once. Glitch cannot guarantee optimal replay orders as it does not maintain a centralized dependency graph to sort over.

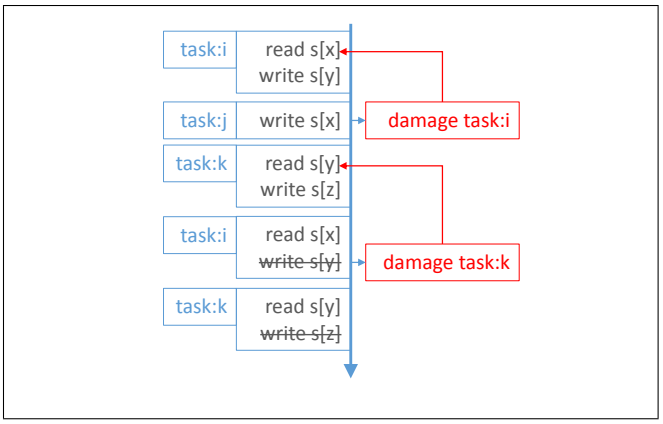

Figure 4: An illustration of how Glitch executes a program; struck through writes are rolled back.

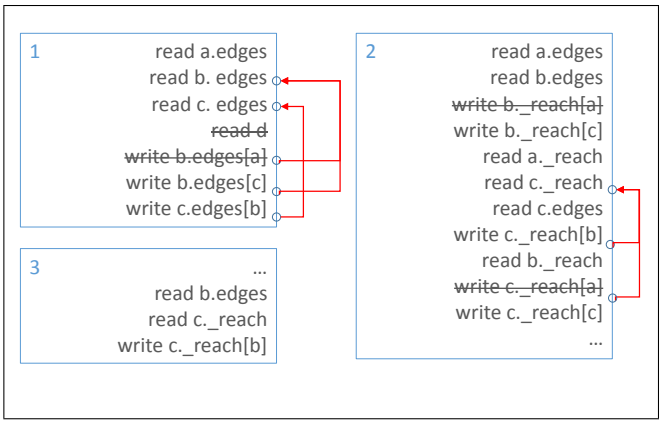

Figure 5: An illustration of how Glitch executes a program with phases (numbered); arrows point to backward dependencies that require a phase increase to read; struck out statements are rolled back after d becomes false.

For simultaneous execution to work, all state updates must be **commutative** so that they can be replayed in arbitrary orders without altering final results. Commutativity is very restrictive: while adding to a set or operating on an accumulator is commutative, cell assignment is not in the general case. Glitch restricts multiple simultaneous assignments to a cell to have equal values. Unequal assignments are flagged as run time errors; consider:

```
cell x, y, z    y = 20    z = 10
x = y + z       z = 22    [re-assign error]
```

Reassignment of z to 10 is presented as an error to the programmer in the editor in the above code. Some reassignments can be resolved gracefully without error; e.g. an event-based discrete state update will have priority over a state update that pre-existed the event, or a default cell assignment in a trait can be subsumed in an extending trait.

The decomposition of a program into tasks does not affect program behavior, only performance. More numerous but smaller tasks can improve incremental performance at the expense of batch performance; e.g. for a compiler, executing the parsing and type checking of all AST nodes in their own tasks leads to a more responsive editing experience as replay is limited to nodes affected by the edit, but it can also lead to slower initial code buffer loads that rely on batch performance. Better incremental performance can be realized if code that likely depends on the same state is clustered into the same task, since such code is likely to be replayed together on a change. For this reason, task decomposition is left to programmers.

State updates, object allocations, and sub-task executions must persist update logs and so are keyed with stable unique *locations* that are reproducible across task replays. A location in YinYang, our language built on top of Glitch, is formed via the memoized lexical token of the expression being executed, changed into a unique path that includes the call stack and loop iterations that the expression is embedded in. These locations are constant across replays even in the presence of adjacent AST deletions and insertions, which is crucial for live execution during code editing.

Taming Cycles with Phases

Dealing with cyclic state dependencies is tricky; consider:

```
L0: object a, b, c are Vertex     L2: b.makeEdge(c)
L1: if d: b.makeEdge(a)           L3: c.makeEdge(b)
```

This example is modified from the Vertex client code in Section 2 to only make a an edge of b if d is true. Suppose that this code initially executes where d is true; a is then in the _reach sets of b and c. Now suppose d somehow becomes false: according to the Task pseudocode in Figure 3, the b.makeEdge(a) call is rolled back, so it is also removed from b._reach; however, a will be re-added as reachable from b on replay of L2 since a is still reachable from c!

This problem is analogous to garbage collecting references that are "stuck" in cycles, which Glitch solves by performing task replay in multiple *phases*. Phases tame cycles by delaying certain state updates to later phases so that tasks do not inadvertently depend on themselves. When an update requires a task to be replayed its phase is set to zero. Execution can only see state updates that occur at or before the current phase; default values are substituted for updates that cannot be seen. State updates made from a statement at a later location than a statement observing that state could indicate a cycle, and so are delayed to a subsequent phase.

Figure 5 illustrates the phased execution assuming in phase 1 that d is true, a and c are added to b's edges set, and b is added to c's edges set (top left of Figure 5). The Vertex constructors that populate vertex _reach sets all execute at L0 for a, b, and c, in that order, so they must wait until phase 2 to add connected members to _reach sets (top right of Figure 5). Since b is constructed before c, c's constructor can see in phase 2 the addition of a and c to b's _reach set. The inclusion of b in c's _reach set is not seen by object b's Vertex constructor until phase 3.

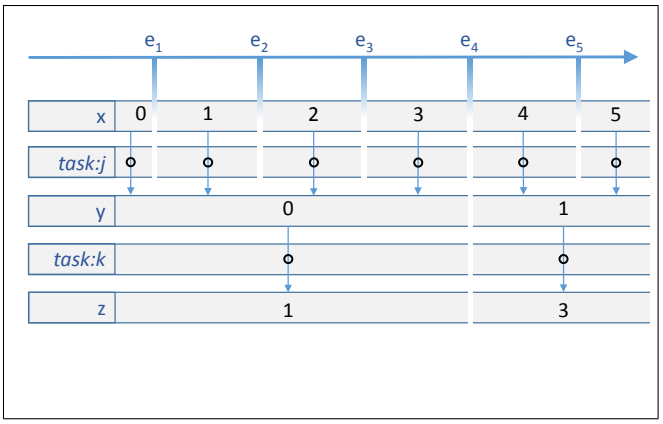

Figure 6: An illustration of how Glitch executes a program over time; arrows point from read to written data; open circles represent task replays.

Now, consider d becoming false, which is illustrated in Figure 5 with struck out reads and writes. Phase 1 replays, removinh a from b's edges set. Phase 2 then replays, removing a from b's _reach set and not re-adding it through a traversal of c's _reach set, which cannot see a there until phase 3 by which time that insertion has been rolled back. Multi-phase replay thus preserves programmability at the expense of replaying tasks for a number of extra phases.

Combining simultaneous execution with cyclic dependency and non-monotonic change permits the encoding of *paradoxes* that will always diverge; consider:

```
set k          if !k.contains(x):   if k.contains(y):
object x, y    | k.insert(y)        | k.insert(x):
```

The not-contains x condition creates a paradox as it guards adding an element that causes itself to become false. Glitch cannot detect paradoxes statically or dynamically and will just replay this code forever in a futile attempt to reach consistency. Paradoxes, like infinite loops, cannot be prevented or detected and so must be debugged manually.

Ripples in the Pond

New state does not just replace old state as a Glitch program changes over time. Program execution and state structures are instead versioned so past execution can be replayed and revised, which is useful in two ways. First, Glitch can speculatively execute while previous executions are still unsettled; e.g. because of uneven workloads, communication delays, or pending IO operations. Second, code can be treated as mutable state to provide better feedback on how edits affect execution; this capability necessarily involves "revising the past," which we discuss later.

State structures maintain time-indexed past histories that can be modified like temporal retroactive data structures [12]. A cell maintains a list of values denoting start and stop times for its assignments along with versioned trees of its readers. Tasks are also split into multiple *traces*, each with their own logs, when there behavior changes over time; updates are coalesced across adjacent traces if they are not rolled back; consider:

```
cell x = 0, y, z
on e:          task: # task:j      task: # task:k
| x = x + 1    | y = (x / 3).floor | z = y * 2 + 1
```

As illustrated in Figure 6, task:j splits and is replayed whenever event e occurs, which causes x to be incremented by 1. However, y is assigned in task:j in a way that does not change as often, so task:k only splits on every third occurrence of e.

Figure 6 illustrates how Glitch avoids replaying execution if dependencies do not change. Glitch performs best on programs that are *temporally coherent* [21], meaning change either does not occur often and/or is quite small; e.g. game entities tend to be stationary or steadily moving—they do not just teleport around! Temporal coherence is high in simulations, games, and most interactive programs.

Glitch can offer better incremental performance by coalescing unchanging state updates and only splitting tasks when they are affected by changes. But this is not enough to offset the cost of replay *bookkeeping* overhead. All reads and writes must be logged for replay and rollback, extra replay phases are added to handle cycles, dynamic type checks must be performed, and method dispatches resolved. These overheads all impact Glitch performance.

There is no way around replaying a task when its dependencies change. However, observe that bookkeeping work really does not need to be redone on a replay whose execution just pushes around slightly different values, which is common in highly temporally coherent programs. Because objects have state with their own dependency queues, assignments to different objects are *big changes* that require redoing bookkeeping work on replay; other big changes include set membership and branching behavior changes. A *small change* is a change in non-object (stateless) values of the same type; e.g. x goes from 3 to 4. Unlike a big change, small change replays can safely reuse previously performed bookkeeping: just the core computation needs to be redone.

As a trace is replayed after a big change, a fast path is computed that can execute more quickly on small changes: all reads and writes are pre-logged and assumed not to change, type errors are pre-accounted for, and dispatches are de-virtualized and inlined. On a small change, this fast path is re-executed quickly, checking only a few invariants such as whether branching behavior remains the same, violations of which trigger a big change. For example in Figure 6, changes to x always cause small changes since it is just changing from one number to another; as a result task:j and task:k are replayed much more quickly after the construction of single traces. Fast paths do however add their own overhead for computation and storage.

A time becomes consistent when all executions up to it have played and no pending changes can affect them. Glitch can then "forget" state and trace history relevant before this time so programs to run for unbounded periods of time.

Live Time Travel

Those who cannot change the past are condemned to start over.

Given Glitch's ability to handle arbitrary changes to a task's behavior, it can incrementally handle code changes while the program is running. As a trace is replayed, the code it executes is traced as a dependency just like the state it reads, allowing for replay when code is edited. Such replay includes traces that executed in the past (if not forgotten) so that an entire program execution can be reactively and incrementally repaired according to how code changes.

The ability to change code in an executing program is an essential part of *live programming*, which, as introduced by Hancock [23], provides programmers continuous feedback on how their code executes as they edit it. Hancock observed that continuous feedback needs a *steady frame* to be useful, meaning (i) relevant variables can be manipulated at locations within the scene (the framing part), and (ii) that variables are constantly present and meaningful (the steady part). While previous work [32] dealt with framing, Glitch addresses the "steady" aspect: given simultaneous execution, each variable has only one observable value between any two events—no stepping context needs to be considered when examining a variable's value.

Glitch can further provide the programming environment with access to the program's history, supporting novel debugging experiences based on time travel, as envisioned by Bret Victor's Learnable Programming essay [48] and his earlier Inventing on Principle talk [47]. A programmer can *scrub* through a program execution, allowing them to inspect its state at different times. For example, a programmer can inspect the trajectory of a bouncing ball by using a slider to visualize the ball at various points in time. A programmer can also visualize multiple times of program execution simultaneously using a film strip of frames [26], or can sample times using *strobing*; e.g. various positions of a bouncing ball can be viewed in the same visualization, giving the programmer a more comprehensive view of how the ball moves over time. Glitch aims to enable these richer programmer experiences in the general case.

4. Experience

Glitch is implemented in C# and can be used as either a library for C# code or can underlay a language like YinYang. Glitch is used as a library in the C# implementation of YinYang's compiler and editor. Code editor lines and blocks are implemented as tasks whose execution determines subtask contents hierarchically. Language AST nodes are also implemented as tasks, providing incremental parsing and semantic analysis "for free" with what otherwise appears as a typical batch-compiler implementation. Glitch's iterative execution allows parsing and semantic analysis to be encoded without explicit multiple passes; e.g. forward symbols references are resolved automatically by replaying code so previously undefined references see symbols when defined.

Our use of Glitch as a library is limited in two ways. First, as explained in Section 3, locations must be provided by C# code in an ad hoc way for use in preserving identity and resolving conflicts. Second, given limitations in the UI framework, all execution is limited to a single time driven by unrecorded inputs that are changed outside of Glitch.

A predecessor to Glitch was conceived to support an interactive Eclipse-based Scala IDE [39]. In this case, one of the authors was able to adapt Martin Odersky's Scala compiler (scalac) in a few months to support managed time by rolling back its non-functional operations, such as symbol table updates that are already commutative. Changes to scalac needed to support managed time were minimal: much of the effort was spent ensuring that values had stable reproducible identities so that non-changes could be recognized as AST nodes were replayed. However some Scala language features were difficult to deal with; e.g. case class constructors are either created explicitly by the programmer or lazily when needed, which was quite difficult to commute. This early experiment with managed time eventually failed for logistical reasons and the current Scala IDE uses laziness instead [38].

Walking the Walk

Embedding Glitch within a procedural language as described above is low-risk: whenever necessary we can escape the confines of managed time back to the chaos of computer time. However such embeddings fail to fully test the limitations of managed time, nor fully reap the benefits of simpler semantics. YinYang, used for the examples in Section 2, is a "pure managed time" language without escape hatches.

YinYang can be used to implement user interfaces, games, and even a compiler, but it is not clear how to express simple things like an in-place bubble sort! Many classic algorithms depend upon re-assignment in one time step and thus do not naturally transliterate well into Glitch; similar limitations occur in single-assignment languages.

Simultaneous execution has significant implications for program and language design that we are still discovering. YinYang avoids many of the conundrums of object constructors, such as when exactly they execute relative to constructors in other classes. Trait constructors can build cyclic structures, and execute along with extending constructors, so they are a great place to maintain or check object invariants. Since trait extension can be rolled back, YinYang can even support *dynamic inheritance* as in Self [45].

For IO, YinYang can replace convoluted callbacks with simple explicit non-blocking control flow; consider:

```
val data = file.read()
if data.exists: ... # will be true eventually
```

This code would typically block until the data is available, so a callback or asynchronous future are often used instead. In contrast, a YinYang read call can return a value that means

"not yet" where the calling task is replayed when the data becomes available in a future time; as with the blocking version, the directness of control flow is maintained. Unlike implicit asynchronous constructs (futures), completion order need not be considered to avoid race conditions.

Performance

Our current work with Glitch focuses on programmability rather than performance: dependency tracing, recording history, state update logging, multi-phase replays, and dynamic code change all have significant performance costs. Initial experience with our prototype suggests that the fine tuning allowed when Glitch is used as a library can scale well; e.g. there are no noticeable problems in the C# implementation of YinYang's programming environment.

We initially found YinYang itself to be very slow: small examples would execute and update quickly, but just a hundred frames of simulating one particle would bring the system to its knees. As mentioned in Section 3, this problem is solvable by compiling traces and replaying them quickly for small changes. Beyond reducing Glitch's overhead, such an approach can improve on dynamic language performance as a form of specialization that is aware of type information. On the other hand, the approach basically compiles everything that is executed for future replays that might not ever come! A comprehensive evaluation of this optimization as well as an exploration of other optimizations is future work.

We take heart from the history of managed memory: garbage collection performance was long a grave concern—it was only when the benefits of managed memory became more widely appreciated that large investments in performance optimization were made, with highly successful results. We claim only that there are many avenues to explore for optimizing the performance of Glitch, and that the benefits of managed time make it worthwhile to do so.

Demos

Short companion demos have been prepared to better describe the live programming experience that Glitch enables; please visit http://bit.ly/1oeuWtB to see them.

5. The Past and Present of Managed Time

It all began with transactions: first in databases [20] and later in programming languages [27]. Transactions isolate asynchronous processes from seeing each other's changes to shared state, preventing them from interfering. Each transaction has it's own time interval in which it has exclusive access to shared state: other transactions are observed occurring either entirely before or entirely afterward. The price for this guarantee is the non-deterministic ordering of transaction execution and the need to sometimes abort and retry transactions. Transactions are proven for isolating independent asynchronous processes, but do not address ordering changes made inside a transaction, nor coordinating multiple causally connected transactions into larger-scale processes.

In Concurrent Revisions [5], state is explicitly shared between tasks that then execute concurrently by "forking" and "joining" revisions. As with transactions, revisions are "isolated:" tasks manipulate their own private copies of shared state that are merged with conflicts being resolved deterministically. Work in [6] puts forth the insight that concurrency, parallelism, and incremental computation share similar task and state decompositions. Cumulative types use commutativity to reduce task conflicts, which are extended in [7] to support distributed computing with eventual consistency.

LVars [29] enforce deterministic concurrency by only allowing monotonically increasing updates to shared state, which are naturally commutative. Interestingly, shared state LVar reads are tamed via thresholds that encapsulate observers from final values that might not have been computed yet. As a result, computations can proceed before final values are available, whereas Glitch must replay computations that read intermediate state values. Bloom [1] likewise relies on enforced monotonicity to achieve "CALM" (consistency as logical monotonicity) where analysis detects non-monotonic program parts that require coordination.

Finally, one of the author's previous work introduces *coherent reaction* [14] where state changes trigger reactions that in turn change other states. Like Glitch, a "coherent execution" order where reactions execute before others affected by their changes is then discovered by detecting incoherencies as they occur and rolling back their updates. Much of the power of imperative programming is also maintained.

We suggest that Concurrent Revisions, LVars, Bloom, Coherence, and Glitch are forms of managed time whose contrasts shed light on the managed-time design space:

- Concurrent revisions "isolate" tasks from each other and merge their effects afterward deterministically;
- LVars enforces strict monotonic updates while providing threshold-based reads;
- Bloom's "CALM consistency" verifies monotonicity to detect where coordination is necessary;
- Coherence's coherent execution that iteratively discovers a coherent statement execution order; and
- Glitch's simultaneous execution that maintains an illusion of order-free execution through replay.

Reactive Programming

Synchronous Reactive Programming (SRP) [2, 9] is inspired by digital circuits whose network of gates apparently execute simultaneously within each clock cycle. Hardware clock cycles are discrete "ticks" of time where interdependent operation results are buffered and fed into the next cycle. SRP originally focused on compilation into formally verified digital circuits, but has been adapted for more general programming [3, 43, 44]. SRP languages tend to restrict the computation within a tick to an acyclic dataflow graph, providing expressions to buffer and control multi-tick computations.

Trellis [13] is a Python library for *automatic callback management* that reactively re-computes specific values. Trellis's abstractions are similar to databinding in production frameworks like Microsoft's WPF [36] and Facebook's React [17] that can encode one-way data flow constraints. Constraint languages like Kaleidoscope [18] are more general performing continuous constraint solving of inequalities. In contrast to these systems, continuous binding is Glitch's default; in fact, special provisions must rather be made using event handlers to prevent continuous replay from occurring.

Functional Reactive Programming (FRP) [16, 24] combines reactive event processing and continuous binding by abstracting time into stream-like *signal* data structures. FRP makes time explicit in the programming model: computations can access the past and time-aware logic like integration can be expressed naively; Glitch programs must remember the past explicitly in event handlers. FRP offers an elegant unification whereby data-processing list comprehensions (i.e. queries) can react to changes via signals; consider:

```
mouseVelBecky u = move offset becky
  where offset = atRate vel u
        vel    = mouseMotion u
```

This code (taken from [15]) causes the pic "Becky" to move with the mouse. Given this dataflow style, all signals must be plumbed centrally; e.g. all movement on Becky must be composed into a single "move" operation. This is also true with Glitch since a position cell could only be assigned once. However, Glitch also supports accumulator and set state constructs, providing more flexibility in how update logic can be diffused in the program. Consider the Particle trait in Figure 2: accumulators sum and count relax call results, which are then used to assign position on each time step. In contrast, since FRP's pureness forgoes the ability to operate on aliases, all relax operations must be composed centrally. Likewise, FRP collections are specified as the output of functions [37] whereas in Glitch they can be built from insert calls on set references (as in Figure 1).

The FRP paradigm focuses on dataflow: rather than "handle" events with additional control flow, one instead composes behavior and event signals. Rx [33] focuses on composing similar event streams in otherwise imperative languages, and there are also many visual languages like VVVV [34] based on reactive dataflow as well. Dataflow is not a clear win: while some composition tasks can become easier, the hidden control flow leads to a very different debugging experience! Glitch's focus on direct control flow is both familiar and very debuggable but lacks composability of event logic, relying on control flow instead.

Push-based FRP systems like FrTime [10], Flapjax [35], and Frappé [11] implement change propagation via a dependency graph that is updated in a topological order to avoid "glitches" where one node update can view state that is updated by a later node update; the node then has to be updated again to account for this inconsistency. Glitch does not try to avoid intermediate glitches, which are in fact unavoidable given cyclic dependencies that are not expressible in FRP. However, Glitch will replay computations until all glitches are flushed out assuming divergence does not occur.

FRP does not support cyclic dependencies because pure functional code is naturally acyclic. In contrast, it is easy to create cyclic dependencies in Glitch since it is based on imperative programming: aliasing references can be mixed up in an imperative operation like cell assignment or set insertion. As an advantage, cycles must be dealt with so Glitch automatically performs iterative processing as part of its semantics, easing iterative tasks such as computing graph vertex reachability (Figure 1). However, expensive multi-phase logic is required to handle non-monotonic changes, while programmers can easily encode paradoxes (`[x = !x]`) directly or indirectly, causing divergence.

Virtual Time

The analogy between managed memory and time is not entirely new; it was made in Flapjax [35] as "consistency as analogous to garbage collection." Grossman [22] explains how transactional memory is to shared-memory concurrency as garbage collection is to memory management; e.g. manual approaches in both are not just hard, but unmodular.

But in fact, Jefferson proposed *virtual time* [25] back in 1985 as an analogue to virtual memory with respect to causally-connected distributed time in order to coordinate distributed processes in discrete event simulation. Virtual time is implemented with an optimistic *Time Warp* mechanism that processes messages as soon as possible, independent of any message that might arrive in the future. Rollback is used to recover from inconsistency that arises as messages arrive out-of-order, where *anti-messages* are used to undo messages that were done by the roll backed computation.

Glitch is heavily inspired by the Time Warp mechanism: tasks are executed as soon as possible without regard to missed future state updates, and rollback recovers from inconsistency. Unlike Time Warp, Glitch does not rollback computations when inconsistency is detected: instead, updates are rolled back only after they are no longer done after replay. Glitch's time-versioned state structures are also inspired by Fujimoto's space-time memory that allows Time Warp to leverage shared memory [19].

Live Programming

Live programming presents a rich programming experience that enhances the feedback between the programmer and executing program [23]. Previous work by one of the author's observed that support for consistent change in an FRP-like programming language is very useful in supporting live programming experiences [31], although dataflow limited its expressiveness, partially inspiring this work. Work in [8] accomplishes live programming for a system where state is externalized as inspired by stateless immediate-mode renders used in high performance graphics; only the present UI state

is kept consistent in response to a code edit, while the model state of the program is not repairable.

To fully realize live programming, we must solve the problem of time. Indeed the current examples of live programming are all cases with a narrow time gap: today we know how to do live programming with pure functions on primitive values as in spreadsheets [42, 50], through dataflow [31, 34], or at the top level of domain-specific programs like games and graphics [8, 23].

Time-travel features have been added to conventional language runtimes [4, 28, 41]. Omniscient debugging [40] relies on vast computing resources to trace and record entire program executions, which is too expensive for casual use. Focusing on object histories alone [30] reduces resource usage, but does not allow for complete time travel. Such facilities are incrementally useful but do not go far enough to support comprehensively live programming. We propose managed time as a fundamental change to programming language semantics that enables fully live programming.

6. The Future of Managed Time

There is still much to be done in the following areas:

- **Usability.** Current managed-time systems often have complex semantics and ask for much programmer discipline. For managed time to succeed, systems must be easy for programmers to understand and use.
- **Expressiveness.** Programmers must be able to write the programs they want to write without hackery; e.g. how would one write a code editor with FRP? Glitch moves in this direction by supporting imperative operations, but is still not fully expressive; e.g. it cannot express reassignment without time to encode algorithms like bubblesort.
- **Performance.** Early garbage collectors were too slow for many, but eventually became viable as technology improved. Likewise, the performance gap between managed and unmanaged time systems will be reduced if adopted. Our attempts in optimizing Glitch are quite early.

There are many tradeoffs to be made; e.g. Concurrent Revisions [5] delivers good performance but requires programmers to adapt to its design constraints, while Glitch trades performance for programmability.

Given their ability to automatically coordinate state updates, managed time systems such as LVars [29] are naturally parallel; Glitch is no exception but we have yet to explore this aspect in depth. In particular, programs in Glitch could be executed in an optimistically parallel manner where unfounded optimism is dealt with through replay and roll back. The ability to work with eventual consistency also aides in efficient distributed computing where network delays make atomicity expensive, as shown by Concurrent Revisions [7], Bloom [1], and Virtual Time [25]. Glitch could also replay tasks on multiple nodes, propagating state updates between nodes so that consistency is achieved eventually.

Glitch treats time as linear, which is perhaps too limited. Conversational programming [42] presents ideas on how debugging can provide programmers with feedback on the program they "could" write will execute, rather than the just the program they wrote already. Similarly, Warth et al's worlds construct [49] allows programmers to control the scope of side effects, allowing them to evaluated multiple possible worlds, throwing away ones found unwanted. To support such systems, managed time systems like Glitch can go beyond time travel to include branching speculative time, allowing programmers to explore multiple possible realities for their programs at once.

Concluding Remarks

The moral of our story is that time is of the essence in a large set of difficult programming issues. Many approaches have been explored in the past and new ones continue to emerge, yet there is still no clear winner. We offer our perspective as a map of the explored regions of alternative models of time, pointing us toward the even larger unexplored regions where new and better solutions may be found. We propose Glitch as a case in point, and look forward to hearing reports from others exploring this frontier.

References

[1] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[2] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *CMU Seminar on Concurrency*, pages 389–448, 1985.

[3] G. Berry and M. Serrano. Hop and hiphop: Multitier web orchestration. In *Proc. of ICDCIT*, pages 1–13, 2014.

[4] S. P. Booth and S. B. Jones. Walk backwards to happiness - debugging by time travel. In *Proc. of Automated and Algorithmic Debugging*, pages 171–183, 1997.

[5] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proc. of OOPSLA*, pages 691–707, 2010.

[6] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation. In *Proc. of OOPSLA*, pages 427–444, 2011.

[7] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proc. of ECOOP*, pages 283–307, 2012.

[8] S. Burckhardt, M. Fahndrich, P. de Halleux, J. Kato, S. McDirmid, M. Moskal, and N. Tillmann. It's alive! Continuous feedback in UI programming. In *Proc. of PLDI*, 2013.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proc. of POPL*, pages 178–188, 1987.

[10] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proc. of ESOP*, pages 294–308, 2006.

[11] A. Courtney. Frappé: Functional reactive programming in Java. In *PADL*, pages 29–44, 2001.

[12] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures. *ACM TALG*, May 2007.

[13] P. J. Eby. Trellis. http://pypi.python.org/pypi/Trellis, 2009.

[14] J. Edwards. Coherent reaction. In *Proc. of Onward!*, pages 925–932, 2009.

[15] C. Elliott. Composing reactive animations. http://conal.net/fran/tutorial.htm, 1998.

[16] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of ICFP*, pages 263–273, 1997.

[17] Facebook. React: A javascript library for building user interfaces. http://facebook.github.io/react, 2014.

[18] B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *Proc. of OOPSLA/ECOOP*, pages 77–88, 1990.

[19] R. M. Fujimoto. The virtual time machine. In *Proc. of SPAA*, pages 199–208, 1989.

[20] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *VLDB*, pages 144–154, 1981.

[21] E. Gröller and W. Purgathofer. Coherence in computer graphics. Technical report, Vienna University of Technology, 1992.

[22] D. Grossman. The transactional memory / garbage collection analogy. In *Proc. of OOPSLA*, pages 695–706, 2007.

[23] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, MIT, 2003.

[24] P. Hudak. Principles of functional reactive programming. *ACM SIGSOFT Software Engineering Notes*, 25(1), 2000.

[25] D. R. Jefferson. Virtual time. *ACM TOPLAS*, 7(3):404–425, July 1985.

[26] J. Kato, S. McDirmid, and X. Cao. Dejavu: integrated support for developing interactive camera-based programs. In *Proc. of UIST*, pages 189–196, 2012.

[27] T. Knight. An architecture for mostly functional languages. In *Proc. of ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

[28] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), Dec. 2003.

[29] L. Kuper and R. R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *Proc. of FHPC*, 2013.

[30] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *Proc. of ECOOP*, pages 592–615, 2008.

[31] S. McDirmid. Living it up with a live programming language. In *Proc. of OOPSLA Onward!*, pages 623–638, October 2007.

[32] S. McDirmid. Usable live programming. In *Proc. of SPLASH Onward!*, Oct. 2013.

[33] E. Meijer. Your mouse is your database. *ACM Queue*, 10(3), 2012.

[34] Meso group. VVVV - a multipurpose toolkit. http://www.vvvv.org, 2009.

[35] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proc. of OOPSLA*, pages 1–20, 2009.

[36] A. Nathan. *Windows Presentation Foundation Unleashed (WPF) (Unleashed)*. Sams, 2006.

[37] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proc. of Haskell*, pages 51–64, Oct. 2002.

[38] M. Odersky. personal communication, 2011.

[39] M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. of OOPSLA*, pages 41–57, 2005.

[40] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. of OOPSLA*, pages 535–552, 2007.

[41] S. P. Reiss. Graphical program development with pecan program development systems. In *Proc. of PSDE*, pages 30–41, 1984.

[42] A. Repenning. Conversational programming: Exploring interactive program analysis. In *Proc. of Onward!*, pages 63–74, 2013.

[43] F. Sant'Anna and R. Ierusalimschy. LuaGravity, a reactive language based on implicit invocation. In *Proc. of SBLP*, pages 89–102, 2009.

[44] F. Sant'Anna et al. Safe system-level concurrency on resource-constrained nodes. In *Proc. of SenSys*, 2013.

[45] D. Ungar and R. B. Smith. Self: the power of simplicity. In *Proc. of OOPSLA*, pages 227–242, December 1987.

[46] G. van Rossum. The Python programming language manual. http://www.python.org, 1990–2013.

[47] B. Victor. Inventing on principle. Invited talk at CUSEC, Jan. 2012.

[48] B. Victor. Learnable programming. http://worrydream.com/LearnableProgramming, Sept. 2012.

[49] A. Warth, Y. Ohshima, T. Kaehler, and A. C. Kay. Worlds: Controlling the scope of side effects. In *Proc. of ECOOP*, pages 179–203, 2011.

[50] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proc. of CHI*, pages 258–265, 1997.