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

Time is of the essence when modifying state. Most programming languages expose the hardware's ability to update global state at any time, leaving the burden entirely on programmers to properly order all updates. Just as languages now **manage memory** to free us from meticulously allocating and freeing memory, languages should also **manage time** to take care of properly ordering state updates. We propose time management as a general language feature to relate various prior work but also to guide future research of this largely unexplored design space.

We propose a new form of managed time, *Glitch*, which ensures that all updates associated with an external event appear to execute simultaneously, removing the need to order execution manually. Glitch **replays** code as needed to reach an appearance of simultaneous execution. To do this, all updates must be commutative and capable of being rolled back, which is ensured through built-in state constructs and runtime monitoring. While these restrictions might seem onerous, we find them acceptable for many realistic programs. Glitch retains the familiar imperative programming model as much as possible, restricting rather than replacing it.

Managed time also aides in *live programming* that provides immediate programmer feedback on how code edits affect execution. Live programming to date has been limited to special cases like pure functions, spreadsheets, or at the top level of games and graphics. Glitch is fully live: past program executions can be replayed in an IDE, while executions are incrementally revised under arbitrary code changes.

1. Introduction

Computers present us with a bleak experience of time: any memory address can be written at any time, determined only by a single global control flow built from branches and calls. But control flow is also used to modularize code into procedures, so writes by modules that affect others (side-effects) are a big problem, made only more dire with the inclusion of concurrency and non-determinism.

Because dealing with state is essential if tedious, many languages attempt to temper it; e.g. state is often encapsulated in objects, and state manipulation can be made explicit with monads [40]. We propose that programming languages also address these problems by abstracting away from the computer's model of time; i.e. the language should **manage time**. We draw an analogy with **managed memory**: it is now widely accepted that languages should unburden programmers from the inhumanly complex problem of correctly allocating and freeing memory. We propose that languages should likewise unburden the programmer from the inhumanly complex problem of correctly ordering state updates.

The concept of managed time serves to relate and contrast many past and ongoing efforts, and also provides a useful perspective from which to assess an entire design space of possible solutions and suggest future research. Time is also a fundamental problem for attempts to improve the programming experience via *live programming* that provides the programmer with continuous feedback about how code executes [18]. Realizing live programming beyond select tame examples will require language management of time.

This paper proposes a new form of managed time, Glitch, which presents the illusion that state updates occur simultaneously even if they are interdependent. The major design goal of Glitch is **programmability**: it retains the familiar imperative programming model as much as possible, restricting it rather than replacing it, while programmers are also free to encode state update cycles as well as changes that are not monotonic. Glitch enables live programming with time travel to past executions and incremental repair of program executions in response to code edits. These capabilities are available for all programs: Glitch is fully live.

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) [14], Concurrent Revisions [5], Bloom [1], and LVars [23] tame state

[Copyright notice will appear here once 'preprint' option is removed.]

```
trait Vertex:
  bag _reach, _connected
  for v in _connected:
    _reach.Add(v)
    for w in v._reach:
      _reach.Add(w)
  def AddV(v):
    _connected.Add(v)
```

Figure 1: The Vertex trait.

update with various techniques (reactive dataflow signals, isolation, strict monotonicity), making different tradeoffs. Section 6 concludes with a future of managed time.

2. Glitch by Example

Consider the following three columns of code written in a language based on Glitch with a Python-like [37] syntax:

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

Cells are built-in Glitch state constructs that can be assigned to arbitrary values. Glitch executes all statements with the appearance of **simultaneous execution**, meaning their order is unimportant and they can see all state updates made by each other. 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. All computations that depend on changing state are updated at the same time that the change occurs. In our example, z becomes 10 when event e fires, causing x to simultaneously change from 42 to 30.

Glitch allows state definition and update to be abstracted into procedures and objects; consider:

```
trait Temperature:
  cell _value
  def Fahrenheit: _value
  def Celsius: 5 * (_value - 32) / 9
  def SetFahrenheit(v): _value = v
  def SetCelsius(v): _value = 9 * (v + 32) / 5
```

In this example, the Temperature trait (a mixin-like class as in Scala [30]) stores a temperature that can be read or set in either Fahrenheit or Celsius values. Reads and updates to a temperature object's state, consisting of a private _value[1] cell, are simultaneous as in the last example:

```
cell x, y, z                  y = 22
object tem is Temperature     z = 50
x = tem.Celsius + y           tem.SetFahrenheit(z - 10)
```

This code's execution assigns x to 26.44. If z later changes from 50 to 70, x would become 42.55 at the same time.

Procedural abstraction of state and their updates is a significant difference between Glitch and dataflow-centric approaches like FRP [14], with important modularity consequences. State definition and updates can safely be encapsulated in modules where the semantics of simultaneous execution removes the need to worry about side-effect ordering between modules. Consider the definition of a Vertex trait in Figure 1. Bags, like cells, are built-in state constructs for expressing unordered set-like collections. The statements in a body of a trait form its constructor, which executes simultaneously with the constructor of any object that extends the trait. Object constructors also execute simultaneously with methods called on the object, and so can observe how the object is manipulated; e.g. Vertex's constructor will see all state updates performed by AddV method calls.

The _reach bag of the Vertex trait contains all vertices connected to connected vertices by any number of hops; i.e. the transitive closure of the _connected relation. Normally calculating a transitive closure in the presences of cycles requires a tricky work-list algorithm. Glitch simplifies this into a simple loop that adds the reachable vertices of all connected vertices. Glitch automatically iterates this calculation to reach a fixed point at the transitive closure. For example:

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

Glitch will replay this code three times given the statement order and the cycle between vertices b and c: on the first replay, the _connected bags of each vertex are populated; on the second, a and c are added to b's _reach bag, while b is added to c's _reach bag; and on the third, b is added to b's _reach bag, while c and a are added to c's _reach bag. Glitch allows cyclic dependencies for improved programmability with much of its technical complexity going to making this work with changes like retracted state updates (Section 3).

Tick Tock Goes the Clock

Time in Glitch consists of *epochs* defined as the time between successive external discrete events; e.g. when a key is pressed, a timer expires, or a request is received. Event handlers execute differently from non-handler code in two ways: they only observe the state of the epoch prior to the event, and their updates are only visible in the epochs that follow the event. Thus a handler's execution is not influenced by any state updates occurring after its epoch, including any performed by itself. Consider Figure 2's Particle trait that extends objects with a physical simulation via a simple Verlet integrator, which uses the last two positions (position and _last) to compute a position _Next with an implicit velocity. Particle constraints are expressed simply as to positions in the Relax method that are interpolated from the implicit next position via a strength parameter; the interpolated positions are then accumulated in the _relaxed accumulator and counted by the _count accumulator, allowing for the particle to be constrained by multiple relax calls. Like cells and bags, accumulators are built-in state constructs that support summation where only the final summed value can be read and

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

```
trait Particle:
  cell position, _last
  accum _relaxed, _count
  def _Next: 2 * position - _last
  def Relax(to, strength):
    _relaxed += _Next * (1 - strength) + to * strength
    _count += 1
  event step
  on step:
    if _count == 0: position = _Next
    else: position = _relaxed / _count
    _last = position   # comment: save position for use in next step
```

Figure 2: The Particle trait.

whose += and -= aggregation operations that are not only commutative, but also reversible.

Event handlers in this code are encoded using the on statement. Particle handles step events by assigning the particle's position to a sum of its relaxed positions averaged using _count; if none, the _Next position is used directly. The particle position of the preceding epoch must also be saved in the _last cell so it can be used in the epoch that follows the step event. The Particle trait can then be used as follows:

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

on Tick:
  fire a.step
  fire b.step
```

This code creates two particles a and b, where a moves to position (5,5) while b both chases a and moves to (0,0). Both objects' step events are then driven by an external clock (Tick) using fire syntax.

Event handlers can specify code that is subject to simultaneous execution in the epochs that follow an event; consider:

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

This code implements a typical UI widget mouse drag adapter that is resilient to mouse resolution errors by using 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 var variables. A continuous future-influenced execution is then specified as an after block where the position of the widget is assigned to its original position plus the delta of where the mouse is now compared to when the mouse down event occurred; the widget now moves with the mouse! Finally, when a mouse up event occurs on the widget, this after block is stopped using a break statement. However, break stops all behavior of an after block, including the widget position assignment! To prevent the widget from reverting to its pre-mouse down position, its position must be assigned to itself in the mouse up handler, which can both "see" the dragged position and whose updates are seen following the mouse up event.

```
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.

3. Making a Time Machine

Glitch's simultaneous execution is a comforting 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. The pseudocode for task replay is shown in Figure 3. When a task is replayed (Replay), all of its existing updates are marked as *stale*. During task execution (Exec), updates are made through calls to Update, which either unmarks the update as stale if it already exists, or installs it if it is otherwise fresh. If an update is still marked as stale in Replay after Exec finishes, the update is no longer performed and so is *rolled back*. Rollback is update specific; e.g. an accumulator operation is reversed; a cell assignment reverts the cell to a default value, and the element added by a bag addition is removed. During execution, a task is added as a dependency to any state it reads (state.depends) so it can be *damaged* and scheduled for repair when an update associated with the state is either 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.

Glitch can replay tasks in non-optimal orders that require state roll back and extra replay; consider:

```
bag b          task:                  task:
object x, y      if !b.Contains(x):     b.Add(x)
                   b.Add(y)
```

Tasks are defined here in `task` blocks. Assuming second-column task is played first, bag `b` will not contains `x` and so `y` will be added to it. Later, the third-column task plays, adding `x` to `b` to damage the second-column task. When the second-column task is replayed, `b` now contains `x` and `b.Add(y)` becomes stale to be rolled back as per Figure 3's pseudocode. Glitch does not attempt to find optimal replay orders, which requires performing an expensive topological sort of a dependency graph that cannot deal with cycles.

For the magic of simultaneous execution to work, all state updates must be **commutative** so that they can be replayed in arbitrary orders without altering final behavior. Commutativity is a strong restriction: cell assignment is not commutative, while operations like adding to a bag are only commutative if remove operations are not supported. Glitch allows such non-commutative operations under the proviso that they do not conflict within an epoch; e.g. a cell can only be assigned to one value per epoch. Unresolvable conflicting state updates 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 conflicts can be resolved gracefully using a priority mechanism; e.g. a state update that occurs because of an event 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 are more sensitive to 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. Glitch leaves performance tuning through task decomposition to the programmer; consider changes to Figure 1's `Vertex` trait:

```
trait Vertex:
  bag _reach, _connected
  task:
    for v in _connected:
      _reach.Add(v)
      for w in v._reach:
        _reach.Add(w)
  def AddV(v):
    _connected.Add(v)
```

`Vertex` is modified from Section 2 so that `Vertex` constructors executes in their own `task` blocks; it then is only replayed on a vertex when its connection bag or the reach bag of a connected vertex changes.

State updates not only alter values but also can create *object identities* for object allocations and sub-task executions that must persist update logs. The identity of an object or sub-task must be keyed and memoized with stable unique *locations* that are reproducible across replays of the same execution. 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.

Some state updates like cell assignment are made commutative through do-once semantics that are enforced dynamically. Totally-ordered locations are also useful here: given two conflicting updates, the earlier update location-wise "wins" leaving the error with the latter; otherwise attribution would be non-deterministic.

Taming Cycles with Phases

Dealing with cyclic state dependencies is tricky; consider:

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

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

This problem is analogous to garbage collecting cyclic references. Glitch replays tasks until a fix-point is reached so all state updates are seen by all statement executions in what is like *hill climbing*, which works well if state updates are either not dependent in cycles or *monotonic*, meaning all updates travel in one direction; e.g. items are only added to a bag and never removed. However, enforcing either invariant in Glitch is too severe; e.g. cyclic graphs could not be processed and connections could never be removed.

Glitch solves the problem of state getting stuck in cyclic update references by performing task replay in multiple *phases*. A phase exists in a time-like dimension that is orthogonal to, and occurring within, epoch time. Phases tame cycles by "delaying" certain state updates to occurring in later phases when cycles could be formed with earlier updates. When a task is damaged, its phase is set to zero. On each replay, only updates that occurred in the current phase can be rolled back as stale. Execution can only "see" state updates that occur at or before; default values are substituted

for state updates that cannot be seen. State updates that occur location-wise after a referencing statement could indicate a cycle, and so a delay is added to when they can be seen.

Using multi-phase replay in this subsection's example, if d is true in phase 0, then a and c are added to b's _reach , and a, b, c are added to c's _reach bag, in phase 0; the AddV call at L2 must replay in a second phase since L2 precedes L3, adding b to its own _reach bag in phase 1. Now suppose d becomes phase and the AddV call at L1 is rolled back, removing a from b's _reach bag. When L2 replays, it does not add a back to b's _reach bag in phase 0, since L2 precedes L3, and so when L3 replays, a is no longer in b's _reach bag, allowing the addition of a to c's own _reach bag to become stale and rolled back. Multi-phase replay is expensive as tasks are replayed for a number of phases related to how updates are related. This tradeoff preserves programmability, while the use of totally-ordered locations reduces when phase delays are added.

Combining simultaneous execution with cyclic dependency and non-monotonic change permits *paradoxes* to arise that prevent fix points from ever being reached; consider:

```
bag k          if !k.Contains(x):    if k.Contains(y):
object x, y      k.Add(y)              k.Add(x):
```

The not-contains x condition leads to a paradox since it guards adding an element that causes itself to become false. Glitch is unable to detect paradoxes statically or dynamically and will replay this code forever, being unable to reach a fix point where all state updates are seen simultaneously. Paradoxes, like runaway recursion or infinite loops, can not be prevented by the language, and like them must be debugged by the programmer.

Change as Ripples in a Pond

Glitch is capable of executing tasks out of order with respect to the epochs they are active in. Out of time execution might occur because a pending event prevents a previous time epoch from yet being consistent, or as we discuss later, because code changes are made that influence past execution. Glitch must then be able to update past program execution, which is achieved in two ways. First, state structures maintain time-indexed versions where their past histories, which can be modified as kinds of temporal retroactive data structures [12]. Second, tasks split into *segments* when behavior must change over time; segments are replayed independently, acting as their own dependencies and maintaining their own update logs. A task initially starts with one segment, which is successively split at epochs where they handle events or their referenced state changes. State updates are coalesced across multiple adjacent segments when possible. For example, a task with the statement `x = (y / 10).Round` splits whenever y changes, but the assignment can change less often as y is divided and rounded; other tasks that depend on x may then split less often.

Glitch's time support is based on two assumptions borrowed from physics and computer graphics:

- Changes are often sparse in occurrence; and
- Changes are often limited in influence.

These principles assume that a program has a high amount of *temporal coherence* in its behavior, meaning change is rare and uniform. By coalescing unchanging state updates and only splitting tasks segments when they are affected by changes, Glitch can potentially offer better incremental replay performance. Glitch is currently unable to exploit non-discrete forms of temporal coherence; e.g. a particle's position changes only a little bit on each step of a simulation, but all segments that depend on this position must be replayed and split regardless. Without temporal coherence, small *butterfly effect* changes can lead to vastly different program behavior. We do not believe that butterfly effect changes arise often in practice, but more study on the topic is needed.

An epoch becomes consistent when no segments that start before it's end need to be replayed and no pending changes can affect its past. In this case, Glitch can *forget* history (task segments and state versions) whose temporal scope ends by the consistent epoch. History can be forgotten once epochs reach consistency but this optimization can be disabled to support time-traveling debugging as described next.

Code Mutation and Time Travel

Given Glitch's ability to handle arbitrary changes to a task's behavior, it can work even when code changes while the program is running. As a task segment is replayed, the code it executes is traced as a dependency just like the state it references, allowing for replay when code is changed in an editor. If past execution history is not forgotten, such replay includes task segments that executed in the past so that an entire program execution can be reactively and incrementally repaired according to how code changes.

The ability to change code while a program is executing is an essential part of *live programming*, which provides programmers continuous feedback on how their code executes as they edit it [18]. Glitch further provides 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 [39] and his earlier Inventing on Principle talk [38]. A programmer can *scrub* through a program execution, allowing them to inspect its state at different time epochs. 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 epochs of program execution simultaneously using a film strip of frames, or can sample epochs 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 Glitch tasks whose execution lays them out hierarchically and determines their contents. 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 implicit iterative execution allows parsing and semantic analysis to be implemented simply as a single pass; for example forward symbols references are resolved automatically by automatically replaying their dependencies when the symbol is 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 epoch driven by inputs from outside the library.

A predecessor to Glitch was conceived in 2007 to support an interactive Eclipse-based Scala IDE [30]. 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. The changes necessary to scalac 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. For example Scala case class constructors are either created explicitly by the programmer or lazily when needed, which required various hacks to deal with the time-sensitive laziness. This early experiment with managed time eventually failed for logistical reasons and the current Scala IDE instead uses laziness [29].

Walking the Walk

Embedding Glitch within a procedural language as described above is low-risk, as whenever necessary one can escape the confines of managed time back to the chaos of hardware time. Such embeddings fail to fully test the limitations of managed time, nor fully reap the benefits of the 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 an in-place bubble sort or an automatically sorted list. Many classic algorithms depend upon multiple assignment and thus do not naturally transliterate into Glitch. Similar limitations occur in pure languages where reassignment is avoided.

Simultaneous execution has significant implications for program and language design which we are only still discovering. YinYang avoids many of the conundrums of object constructors, such as when exactly they execute relative to constructors in other classes. YinYang constructors can build cyclic structures. Trait constructors in YinYang execute along with extending constructors, so they are a great place to maintain or check object invariants. Trait extension can change dynamically, supporting *dynamic inheritance* as in Self [36], which is feasible because constructors execute simultaneously, avoiding tricky initialization issues.

YinYang allows the use of simple explicit control flow rather than convoluted callbacks or asynchronous execution constructs. Consider:

```
val data = file.Read()
```

In a typical imperative language, this code will block until the data is available. We could also install a callback to be called when the data is available, doing other work in the meantime, at the expense of having code called sometime in an undetermined future. A future that acts as a handle to the result can be used, but the computation may still block when the future is thunked. In contrast, in YinYang, the Read call returns a value that means "not yet" where the calling task is replayed when the data becomes available in a future epoch; as with the blocking version, the directness of control flow is maintained. Unlike implicitly asynchronous language constructs, the programmer need not consider what the actual execution order of events will be to avoid creating race conditions. Likewise event handling occurs via control flow in YinYang rather than being transformed into dataflow streams as occurs in FRP [14] and other FRP-like systems.

Performance Deferred

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.

On the other hand YinYang itself is currently slow: while small examples execute and update quickly, just a hundred frames of a physics animation can bring the system to its knees. Currently, the most expedient way to improve performance is to port lower level traits (like Particle) to C# and fine tune their replay; e.g. to use imperative sequencing when locally feasible to avoid the expense of tracing and logging. Implementation optimization of Glitch's runtime support is the subject of 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 for optimizing the performance of Glitch to be explored, and that the benefits of managed time make it worthwhile to do so.

Demos

The following short demos have been prepared to better describe the live programming experience that Glitch enables; please select the "cog" for each video to set the resolution to 360p (otherwise the code is too blurry to read) with annotations turned on. The demos are as follows:

- http://youtu.be/MtYyn_Rt8zk: a simple demo of simultaneous execution using the Temperature trait.
- http://youtu.be/hDoc_62rQE8: a demo of multi-phase replay using the Vertex trait, and what happens when a paradox is accidentally encoded?
- http://youtu.be/xmWJsTTOvkU: a simple physics demo that demos time travel, scrubbing, and strobing.

5. The Past and Present of Managed Time

It all began with transactions: first in databases [17] and later in programming languages [21]. 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 a proven solution for isolating independent asynchronous processes, but do not address the problems of ordering changes made inside a transaction, nor coordinating multiple causally connected transactions into larger-scale processes.

Programmers can use concurrent revisions [5] to declare what state they wish to share between tasks that then execute concurrently by "forking" and "joining" revisions. As with transactions, revisions are "isolated:" tasks read and modify their own private copy of shared state, which are then merged where conflicts are resolved deterministically. This system is enhanced in [6] with the insight that concurrency, parallelism, and incremental computation share similar task and state decompositions. They also introduce cumulative types that use commutativity to reduce conflicts between tasks, and extend this with cloud types [7] to support distributed computing while also making eventual consistency more manageable to programmers.

LVars [23] 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 *language-level consistency*with "CALM consistency" where analysis can detect non-monotonic parts of a program that require coordination.

Finally, one of the author's previous work introduces *coherent reaction* [13] where state changes trigger events called reactions that in turn change other states. A "coherent execution" order, where reactions execute before others affected by their changes, is then discovered iteratively by detecting incoherencies as they occur and rolling back their updates, as Glitch does. As in Glitch, much of the power of imperative programming is maintained.

We suggest that Concurrent Revisions, LVars, Bloom, Coherence, along with Glitch are all forms of managed time. To varying degrees, these systems leverage commutativity, eventual consistency, and monotonicity (enforced or otherwise). The differences between these systems reveal dimensions of 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 abstracted into discrete "ticks" of time in which a network of interdependent operations appears to execute simultaneously, with the results buffered and fed into the next cycle. SRP was originally focused on compilation into digital circuits and formal verification, but has since been adapted for more general software problems [3, 34, 35]. SRP languages tend to restrict the computation within a tick to an acyclic dataflow graph, and provide linguistic expressions to buffer and control multi-tick computations.

Functional Reactive Programming (FRP) [14, 19] abstracts time into stream-like data structures representing either discrete events or continuously changing values that can be observed at discrete times. 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 "signal" constructs. Glitch in contrast is more imperative: state-updating operations can be called anywhere in code even if those updates are limited to being commutative and undoable. Each scheme has different qualities: FRP logic for some state can be composed out of functions into a dataflow network that is bound once, while Glitch diffuse updates throughout the program. The ability to diffuse update logic in a program is good for modularity but comes at the cost of composability.

The FRP paradigm focus primarily on dataflow: rather than "handle" events with additional control flow, one instead composes behavior and event signals. Rx [26] focuses on composing similar event streams in otherwise imperative languages, and there are also many visual languages like VVVV [27] based on reactive dataflow as well. It is not clear that dataflow is a 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.

Push-based FRP systems like FrTime [10], Flapjax [28], 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. To temper cycles, Flapjax requires explicit time delays while Glitch adds time-orthogonal phase delays that are programmer transparent.

Virtual Time

The analogy between managed memory and time is not entirely new; a similar observation was made for Flapjax [28]:

> *We view consistency as analogous to garbage collection: a sensible requirement that is so pervasive that languages should try to support and optimize it.*

In fact, Jefferson proposed *virtual time* [20] 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 then used to recover from inconsistency that can arise 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 [16].

Live Programming

Live programming presents a rich programming experience that enhances the feedback between the programmer and executing program [18]. 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 [25], although dataflow limited its expressiveness; in particular, the implementation of the live editor had to be done in the same proto-Glitch managed-time system that was used by the same author in the Scala IDE work. Work in [8] shows how live programming can be accomplished for a system where state is externalized as inspired by stateless immediate-mode renders used in high performance graphics. In this approach 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 in a general way, we must solve the essential problem of time. Indeed the current examples of live programming are all cases with a narrow time gap: today we only know how to do live programming in special cases, such as pure functions on primitive values as in spreadsheets [33, 41], through dataflow [25, 27], or at the top level of domain-specific programs like games and graphics [8, 18].

Time-travel features have been added to conventional language runtimes [4, 22, 32]. Omniscient debugging [31] relies on vast computing resources to trace and record entire program executions, which is too expensive for casual use. Focusing on object histories alone [24] 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 complicated semantics and ask for much discipline from the programmer; e.g. Glitch has strange latching event handling semantics. For managed time to succeed, systems must be easy for programmers to understand and not ask too much from them.
- **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 the use of time to encode basic algorithms like bubblesort.

- **Performance.** Early garbage collectors were too slow for many applications, but eventually became viable as both hardware and technology improved. The performance gap between managed and unmanaged time systems should be reasonable if these systems are to be adopted. We have not yet attempted to optimize Glitch. While there are many avenues to explore, we must realistically admit that Glitch's sophisticated runtime support has costs that probably cannot be completely eliminated.

There are many tradeoffs between these areas; e.g. Concurrent Revisions [5] delivers good performance but requires much programmer discipline who must carefully fit their programs into its model, while Glitch trades performance for more programmability.

Given their ability to automatically coordinate state updates, managed time systems such as LVars [23] 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 any unfounded optimism can be dealt with through replay and roll back. The ability to work easily with eventual consistency is also essential to efficient distributed computing where network delays make atomicity very expensive, as shown by Concurrent Revisions [7], Bloom [1], and Virtual Time [20]. Glitch could also replay tasks on multiple nodes, propagating state updates between nodes so that consistency is achieved eventually.

In realizing simultaneous execution, Glitch leverages replay to ensure that all statements execute seeing all state updates performed at a given time. Constraint languages like Kaleidoscope [15] achieve a similar outcome with constraint solving, rather than replay, where consistency is defined more generally to include inequalities. Enhancing Glitch to support similar inequalities would be a huge boon to expressiveness. Conversational programming [33] 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; similar ideas were also presented in Victor's learnable programming essay [39]. 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 other explorers of 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 *Seminar on Concurrency, Carnegie-Mellon University*, 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] J. Edwards. Coherent reaction. In *Proc. of Onward!*, pages 925–932, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] Meso group. VVVV - a multipurpose toolkit. `www.vvv.org`, 2009.

[28] 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.

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

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

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

[32] S. P. Reiss. Graphical program development with pecan program development systems. In *Proc. of Practical Software Development Environments*, pages 30–41, 1984.

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

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

[35] F. Sant'Anna et al. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of SenSys'13*. ACM, 2013.

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

[37] G. van Rossum. The Python programming language manual. www.python.org, 1990–2013.

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

[39] B. Victor. Learnable programming. `worrydream.com/LearnableProgramming`, Sept. 2012.

[40] P. Wadler. Comprehending monads. In *Proc. of LISP and Functional Programming*, pages 61–78, 1990.

[41] 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.