Programming With Time

Cyber-physical programming with Impromptu

Andrew Sorensen
Australian National University
andrew@moso.com.au

Henry Gardner
Australian National University
Henry.Gardner@anu.edu.au

Abstract

The act of computer programming is generally considered to be temporally removed from a computer program's execution. In this paper we discuss the idea of programming as an activity that takes place within the temporal bounds of a real-time computational process and its interactions with the physical world. We ground these ideas within the context of *livecoding* – a live audiovisual performance practice. We then describe how the development of the programming environment "Impromptu" has addressed our ideas of programming *with time* and the notion of the programmer as an agent in a cyber-physical system.

1. Introduction

An *act of programming* is usually considered to sit firmly within the context of "software development", with the latter being the active process behind the production of "software products". In this view, causal actions made by the programmer target the design and notation of a formal specification for *future* action by a computing system.

One significant ramification of this traditional view is to promote a strong separation between the *program*, *process* and *task* domains [43]; a program being a static specification of intention (i.e. code), a process being the reification of the program on a given computing machine, and the task domain being a target for real-world affect. This view is so pervasive as to have dominated the historical practice of computer programming.

In this paper we discuss an alternative view, where programming is a causal and ephemeral practice whose action directly targets a physical task domain. This view gives privilege to causal action by projecting the programmer's human agency into the world. The traditional, temporal separation between *program*, *process* and *task* domains, is elided as programmer action is made directly affective in the task domain. The programmer becomes an active agent operating within a cyber-physical feedback system, orchestrating the real-time evolution of computational processes which affect the physical environment. In turn the environmental, task domain provides the programmer with real-time feedback to inform her future actions.

This realignment of programming, from a static to a dynamic activity, requires a suitable realignment to the substrate of programming infrastructure. In particular when shifting programming from an "act of specification" to an "act of action" it is not only the runtime system that is subject to the temporal bounds of the task domain but the programming tools themselves. By placing the programmer as an active agent in a cyber-physical feedback system we are opening the programming task up to a broad range of time and concurrency demands.

Compilers, linkers, analysers and the like are all traditionally considered to be "out of time" processes. However, when integrated into a dynamic, real-time, causal development process, such tools must be reactive, complying to task-domain time and concurrency constraints as well as being amenable to reactive updates from the environment.

In this paper we explore the idea of programming as an act of human agency bounded in temporal alignment with a real-world task domain. We consider the implications of this model for programming language systems and provide concrete illustrations using the programming environment "Impromptu". In the next section we start by considering a reference scenario from a digital-arts practice known as "livecoding" [10] and we show that the key concerns which emerge are those of temporality, scheduling, concurrency and the human computer interface. In Section 3, we then introduce a paradigm which we call "With Time Programming" (WTP). The main technical section of this paper, Section 4, describes the Impromptu WTP environment which has been used for livecoding and live audiovisual performances around the world over the past five years. Finally, Section 5 makes a connection between WTP and programming support for human agency in cyber-physical systems.

2. A Reference Scenario

As discussed above, the realignment of programming as a dynamic activity transforms it from a static specification of intention into a vehicle for direct causal affect. This transition elevates the human programmer's role in the computational process to one of active agency. In this section, a traditional example and a modern scenario from music and multimedia practice are introduced to provide an orientation for the rest of this paper.

Consider the analogy of a traditional musical score. The score provides a static specification of intention – a static program domain. Musicians, representing the process domain, perform the actions required to realise or reify the score. Finally, the actions in the process domain result in sound waves which are perceived by a human audience as music. This final stage is our real-world task domain. Now consider a dynamic program domain in which a composer conceives of and describes a musical score in real-time. We commonly call this type of composition *improvisation*. In it, the improvising musician is involved in a feedback loop involving forethought, moving to causal action and finally to reaction, refinement and reflection.

Livecoding [10, 51] is a computational arts practice that involves the real-time creation of generative audiovisual software for interactive multimedia performance. Commonly the programmers' actions are exposed to the audience by projection of the editing environment. Livecoding performances often involve more than one participant, and are often commenced from a conceptual blank slate [49]. Because of the highly temporal nature of linear media, sound and visual animation, livecoding provides a strong example of what we call "with-time programming". We now present a short story describing a possible livecoding performance:

Two performers are present on stage. One, a violinist, stands paused, bow at the ready. Another sits behind the glow of a laptop screen. A projection of the laptop screen is cast above the stage showing a blank page with a single blinking cursor. The laptop musician begins to type ...

```
(play-sound (now) synth c3 soft minute)
```

... the expression is evaluated and blinks on the overhead projection to display the performer's action. An ethereal synthetic sound immediately enters the space and the violinist begins to improvise in sympathy with the newly evolving synthetic texture. The laptop performer, listens to the thematic material provided by the violinist and begins to outline a generative Markov process to accompany the violin ...

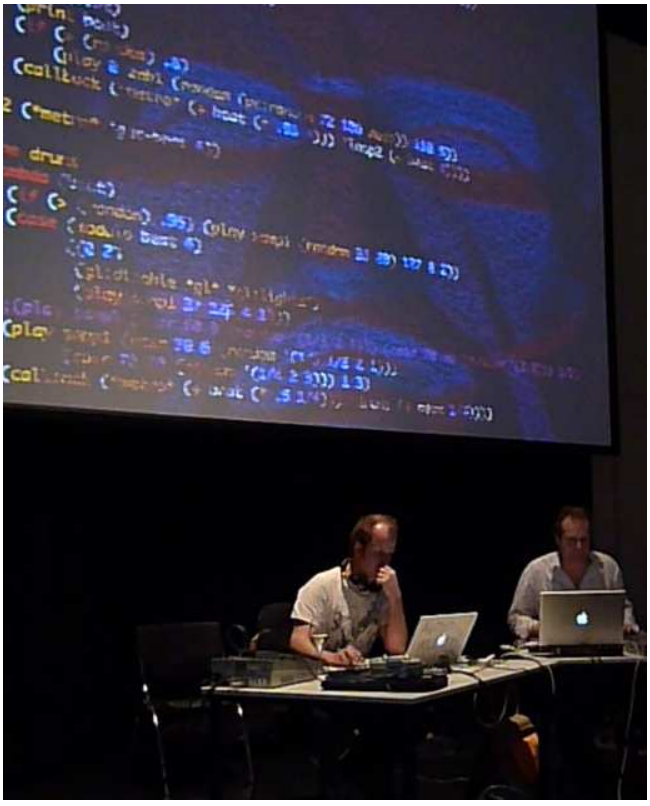

Figure 1. Livecoding Duet (Bencina & Sorensen 2009)

```
(define chords
  (lambda (beat chord duration)
    (for-each (lambda (pitch)
                (play synth pitch soft duration))
              chord)
    (schedule (*metro* (+ beat duration)) chords
              (+ beat duration)
              (random (assoc chord '((Cmin7 Dmin7)
                                     (Dmin7 Cmin7))))
              duration)))

(chords (*metro* 'get-beat 4) Cmin7 4)
```

... the "chords" function is called on the first beat of a new common time bar and a simple recursive chord progression begins supporting the melodic performance of the violin. The chord function loops through time, creating an endless generative sequence of four beat chords. After a few moments of reflection the laptop performer begins to modify the "chords" function to support a more varied chord progression with a randomised rate of temporal recursion ...

```
(define chords
  (lambda (beat chord duration)
    (for-each (lambda (pitch)
                (play dls (+ 60 pitch) soft duration))
              chord)
    (schedule (*metro* (+ beat duration)) chords
              (+ beat duration)
              (random (assoc chord '((Cmin7 Dmin7 Bbmaj)
                                     (Bbmaj Cmin7)
                                     (Dmin7 Cmin7))))
              (random '(3 6)))))
```

... the laptop performer finishes editing the desired changes and pauses to listen to the violinist, waiting for a musically sensitive moment to introduce the new code changes. The code is evaluated and hot-swapped on-the-fly. The chord progression continues on without any noticeable interruption. The violinist responds in kind by improvising over the new progression inspiring the laptop performer to introduce a basso profundo voice ...

```
(define basso
   (lambda (beat pitch duration)
      (play bass pitch soft duration)
      (schedule (*metro* (+ beat duration)) 'basso
                (+ beat duration)
                (if (< pitch 37)
                    48
                    (pc:relative pitch -1
                                 (pc:scale 10 'ionian)))
                duration)))

(basso (*metro* 'get-beat 4) 48 8)
```

... the basso profundo voice enters on the first beat of the next bar, perfectly synchronising to the ongoing chord progression with a slowly falling Ionian scale. In this way a performance unfolds over time, with both performers acting in the world – executing and modifying plans in response to reactive feedback from the environment.

This story illustrates a number of key concerns. The laptop performer is affecting change in-the-world in real-time which is only possible when the program, the process and the task domains are temporally aligned. The livecoding performer must be able to realise updates to the process domain, through the program domain within the temporal bounds of the task domain. In practical terms this means that the system needs to support direct manipulation by the programmer within the bounds of human audio perception – fractions of milliseconds for some audio tasks. The system also needs to support the temporal alignment and scheduling of concurrent processes: both the chords and the basso profundo voices needed to be synchronised temporally and the basso profundo was scheduled for introduction at a time which made sense musically. Finally, the system needs to support multiple human cognitive cycles of attention, planning and action. In particular, there were times when the programmer was planning future action, by building processes that infer future temporal agency. These planning phases occur at the same time that a programmer is also attending to the ongoing performance and they may need to be interrupted to adjust other aspects of the music or visual display. Although planning phases involve limited temporal association between the program and process domains, there is a need for temporal reasoning that is tied to the task domain through real-world, clock-time semantics.

Although this scenario is artificial, the first author of this paper is an active composer and livecoding performer. His work has been performed around the world and is viewable online as recorded video screen-casts [47].

3. With-Time Programming

With-Time programming (WTP) extends ideas of "just in time programming" [37], "experimental programming"[42] and "live programming" [1, 15, 31, 44], to include timing constraints based on real-world clock-time. We claim that a focus on time driven semantics provides programmers with a strong causal connection to the real-time world. Aligning time in the program, process and task domains helps to support human action cycles including planning, execution and feedback. In this section we position With-Time programming with respect to its comparitors.

3.1 Just In Time and Experimental Programming

Just in Time (JIT) programming prescribes that algorithmic description and evaluation occur within a temporal quantum that is appropriate to a given task domain. Richard Potter offers the following description:

> ... the goal of just in time programming is to allow users to profit from their task-time algorithmic insights by programming. Instead of automating with software what was carefully designed and implemented much earlier, the user recognises an algorithm and then creates the software to take advantage of it just before it is needed, hence implementing it just in time.[37]

Potter's emphasis that the "user" as a *spontaneous* algorithm creator is fundamental to JIT programming and this distinguishes it from other design-centred, engineering practices.

JIT programming aligns with an iterative and incremental development cycle common to Agile software development methodologies. Like Agile methods, JIT programming advocates a style of negotiated development between programmers and their work. Agile methods acknowledge the limitations of formal specification in the development of real-world commercial projects and attempt to systematise the development cycle to accommodate unknown, or poorly known, requirements [4]. Where JIT programming diverges from Agile thinking is that it is fundamentally more transient in nature. Not only is development a negotiation, it is also ephemeral. Indeed, during the course of a JIT session, source code which was valid at the start may be re-appropriated for a different purpose at the end. In other words, source code will expand, contract and morph during a given session and, significantly, the source code at any point in time will only provide a partial view of the active runtime system. Today there are many, widely-used JIT programming environments – we mention the R statistical environment as one successful example[39].

For the JIT programmer, there is often no intention to create a final software product and JIT programming is about experiment rather than manufacturing. The term "Experimental Programming" has sometimes been used in the same

sense as JIT programming and it has also been used to denote some specific JIT programming projects. It has a long history in the Lisp community as reflected in the following quote by Erik Sandewall:

> The average Lisp user writes a program as a programming experiment, i.e.., in order to develop the understanding of some task, rather than in expectation of production use of the program.[42].

Seymour Papert attempted to start an education revolution around the idea of experimental programming which resonates to this day in the work of the Lifelong Kindergarten Project at MIT including projects such as Logo [36], Flogo I & II [19] and Scratch [28]. These projects all share an interactive and experiential view of programming, which sees the programmer as an active learner engaged in a computational experiment.

3.2 Live Programming

Live Programming (LP) is a term which has been used to denote systems which support the direct intervention of the programmer in a program's runtime state. It can be thought of as an extreme version of JIT programming where there is a direct correlation between a program's representation and its execution. For example:

> In [SubText] the representation of a program is the same thing as its execution. Aligning syntax and semantics narrows the conceptual gulfs of programming. The experience of programming becomes more akin to using a spreadsheet than a keypunch [15].

Live Programming often shares an interest in real-world experiential and experimental programming practice and often seeks to provide the programmer with a direct connection to the physical world:

> In our vision, the Self programmer lives and acts in a consistent and malleable world, from the concrete motor-sensory, to the abstract, intellectual levels. At the lowest, motor-sensory level of experience, objects provide the foundation for natural interaction. Consequently, every visual element in Self, from the largest window to the smallest triangle is a directly manipulable object. [44]

Examples of LP interfaces maybe text-driven systems where each character that is inserted by a programmer is immediately interpreted and acted upon – even before a command termination character has been received. Some visual-patching environments also support a notion of live programming where on-the-fly changes to a data-flow graph are immediately interpreted. For example, environments such as Max/MSP and PureData, support immediate updates to a signal processing graph in response to changes in the visual programming interface[38].

3.3 Features of WTP

WTP is an extension of JIT programming which emphasises clock-time and interaction with real-world artefacts. WTP is an experimental and experiential practice that focuses on the dynamic negotiation of artefacts that exist in-the-world. It attempts to be incremental and interactive and it supports hot updates to the runtime system. It is reactive to the environment, providing feedback to the programmer both through the user interface and also through the programming language infrastructure. WTP's emphasis on real-world interaction mandates that time and concurrency be first-class concerns of, not only the runtime system, but of the whole programming infrastructure.

As shown in the reference scenario, WTP can support direct and immediate updates to the environment. However, WTP code is a transient interface for intervention rather than being either a static specification, or a live representation. Ultimately, WTP attempts to provide the programmer with a strong causal connection to the world by correlating time and concurrency across the program, process and task domains.

There is at least one reference in the literature where the term Just In Time programming has been used to apply to our conception of WTP: Rohrhuber et al. describe their *JITlib* extension to the SuperCollider environment, which is also in the domain of livecoding of audiovisual media. We will consider this system in Section 4.5 (Related Work).

4. Impromptu

In this section we describe Impromptu, a WTP system which has been progressively developed and applied in the livecoding context over the past five years [45].

As discussed above, a WTP system must address a number of key concerns: It needs to support "first-class" temporal semantics and a natural style of concurrent programming; it needs to provide an integrated, reactive, development environment; and it needs to include adequate real-time systems support for the particular task domain. In the following sections we outline how Impromptu addresses these issues.

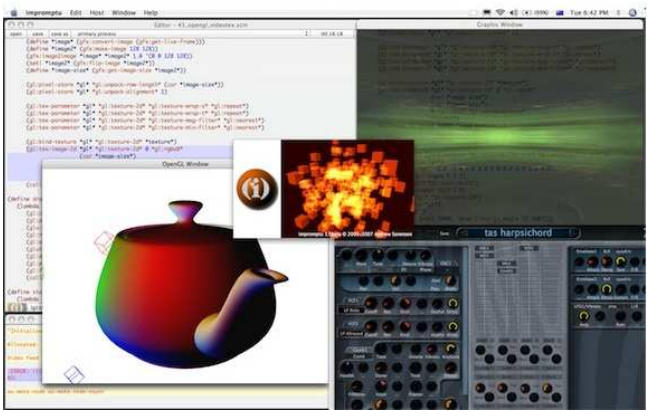

Figure 2. The Impromptu IDE

4.1 Dynamic and reactive infrastructure

The programming language infrastructure of a WTP system must support dynamic updates to the underlying runtime system in a *timely* manner – in other words, the program domain must be able to dynamically influence the process domain. A WTP system must also be reactive in nature, in that it provides suitable real-time feedback to the programmer and, ideally, supports various automated mechanisms for reactively updating the underlying runtime system. Because programming and execution are temporally aligned, these observations imply that WTP development tools must, themselves, be dynamic and reactive: they must comply with time and concurrency constraints from the task domain and need to be amenable to reactive updates from the environment.

Impromptu provides an Integrated Development Environment (IDE) based around a Scheme language implementation and an audiovisual runtime system. The environment includes, a source code editor, a Scheme interpreter [1], an experimental Scheme-to-x86 compiler (via LLVM), a concurrent (incremental) garbage collector, a debugger, an audio DSP architecture and a substantial graphics subsystem. Additionally, Impromptu supports a number of options for extension, including a foreign-function interface, a bidirectional Objective-C bridge and audio and visual plug-in architectures.

Impromptu supports multi-core and distributed processing – possibly from multiple, collaborating programmers. Its user-interface (UI) allows programmers to select an active process and then its editing window is used to plan and enter functions for scheduling and evaluation. By automated selection and highlighting, the editor allows programmers to send arbitrary expressions to an Impromptu process for evaluation – either via interpretation or JIT compilation. Impromptu strives to make this process as timely as possible. In the context of multimedia performance this means being able to, for example, "perform" through the interface, evaluating expressions *in time* to the rhythms of a live ensemble. This is a task that requires co-ordination between the editor, evaluation engine and underlying runtime system.

The Impromptu IDE provides an array of standard text-editing tools such as keyword-completion, syntax highlighting, keyboard shortcuts, expression matching, error highlighting, text macros, undo/redo, search-and-replace, and so on. The editor is also programmable, providing both programmers and the Impromptu system itself with the ability to modify text procedurally. This provides for programmer feedback directly through the text interface.

An example of this textual feedback in action is the case of *autonomous debugging*. Autonomous debugging is an important consideration for WTP as the programmer is often involved in other real-time activities when an error occurs – a debugging task interrupts human activities such as listening, musical planning and orchestration and too much time spent attending to debugging can seriously detract from an overall performance. Ideally we would like the debugger to intercede to automatically solve simple problems and to provide notification and to update the source code to reflect the actions taken. One example where this occurs is in the case of unbound variables in situations where Impromptu's analyser can discern that the unbound variable can be "safely" represented with a suitable type and value ("safely" in a program/process-domain sense rather than a task-domain sense). The debugger can then automatically substitute the value into the abstract syntax tree and update the source code to reflect the change.

There is no doubt that this style of autonomous debugging has a number of task domain ramifications but it is arguably less invasive than automatically interrupting the programmer from her current task to engage in a debugging activity. Maintaining execution of the concurrent process until the programmer finds time to address the error can be a useful tool. By updating the code, and highlighting the change, the programmer is made aware of the modification and can take action when time permits.

Impromptu's programmable editing environment also allows programmers to build autonomous agents that can generate and/or modify the source code at runtime. This style of autonomous editing was a feature of Alex McLean's Perl environment [32] and has recently become a central feature of Thor Magnusons ixi language [27].

The Impromptu UI is responsible for displaying information about the state of the runtime system. Sometimes this information is presented discretely, as with the debugging example above, but there are many situations where it is continuous. This is the case for load information such as memory usage, CPU performance, and, more specifically, the system's ability to meet real-time deadlines. The Impromptu UI provides real-time feedback about the system's ability to meet timing constraints using a graphical, heads-up-display that floats above the source code. Timing of temporal recursions and other system resource information is graphically displayed to the programmer over the text interface.

Additionally, the Impromptu UI provides canvas and plug-in support to the underlying audio and graphics subsystems. Plug-in UIs such as those in Apple's AudioUnit specification, provide real-time gestural control over plug-in parameters and supplement Impromptu's procedural access to these same parameters. (This combination of direct manipulation and procedural control is another example of the coupling of *program* and *process* domains.) The Impromptu graphics canvas provides any number of either OpenGL or Quartz graphics contexts for display on any connected displays, at any given resolution. Impromptu's canvases provide a number of useful abilities such as the ability to capture as an image (PDF or bitmap) or as a quicktime movie,

[1] Impromptu's original interpreter was forked from the TinyScheme project [50]. It has been substantially modified since the original fork.

to write or read directly to or from another canvas (be that OpenGL or Quartz), to capture images directly from the underlying window system, and so on. Finally, the Impromptu Objective-C bridge allows for custom OSX Cocoa UIs to be built on-the-fly and manipulated either procedurally or directly. Impromptu can, and has been, used to develop standalone OSX desktop applications [46].

4.2 Concurrency

In order to support the high degree of parallelism inherent in many real-world task domains, a WTP system should have a concurrency model which is *lightweight* [2]. It should attempt to be syntactically and semantically *simple*. It should be sufficiently *efficient* to support the domain activity and it should be *flexible* to allow for easy integration with third-party libraries and remote systems.

Impromptu responds to the first three requirements by adopting a concurrency design pattern which we have dubbed "temporal recursion" (linguistically rather than mathematically recursive). Temporal recursion is a style of time-driven, discrete-event concurrency [6, 24] similar to *Engines* [20, 21] with real-time interrupt scheduling.

By scheduling future events in a self referential manner, a temporal loop is formed. Temporal recursion provides an asynchronous, deterministic concurrency framework with a simple semantics for both time and concurrency. It is the primary concurrency paradigm in Impromptu and in the simplest case, of non-distributed, single-core operation, it is the only model which programmers need to reason about. The fourth requirement, of flexibility, is implemented using a multi-layer concurrency model as described below.

The temporal recursion model is an extension of Impromptu's support for the real-time execution of arbitrary code blocks such as procedures, closures and continuations. A real-time scheduler is responsible for scheduling the execution of code blocks with chronological ordering. This engine is based on a priority queue containing a dynamic number of *event structures* – tuples containing an execution start-time, a maximum execution run-time, a process identifier, a function (procedure, continuation or closure), and a vector of one or more argument values as required by the given function. Events can then be applied using standard procedure-call semantics by applying the supplied arguments to the given function. (The scheduler protects message tuples from garbage collection). More details on temporal recursion are given in the next subsection.

Impromptu's temporal recursion model and Scheme's top-level interaction work together to support a natural and flexible style of hot-swappable code. The Scheme language makes code updates "natural" in that modifications to the source code are reflected in the next function call – so that dynamic updates from the program domain are reflected in the process domain.

Impromptu supports multi-core and distributed computation by supporting a notion of *processes*. Processes are pre-emptive and each provides an isolated execution stack, heap memory and real-time garbage collector [2]. Interaction with other Impromptu processes is strictly via message passing through Impromptu's inter-process communication (IPC) mechanism.

Impromptu's temporal recursion events specify a process ID and may be directed to execute on any available Impromptu process. The temporal-recursion discrete-event model mandates the sequential execution of functions and each Impromptu process is responsible for executing these functions as fast as possible. Impromptu supports standard Scheme semantics particularly with regard to state semantics. As in standard Scheme, it is possible to program in both functional and imperative styles using the temporal recursion model and this provides a straight-forward progression from standard Scheme programming to Impromptu programming.

A multi-layered approach to concurrency such as this places additional cognitive load on the programmer because he must be able to reason about programs with suitable regard to all layers. However, a multi-layered approach is the most practical solution for any system attempting to provide a deterministic concurrency model while supporting non-deterministic distributed computation and external library integration. Impromptu attempts to balance these needs with a desire to provide a simple, deterministic concurrency model and makes this model explicit. Keeping concurrency and time as explicit concerns is, as we have previously mentioned, an issue that we think is of intrinsic importance to WTP.

4.2.1 Temporal-recursion

In this subsection, we provide concrete examples of Impromptu's temporal-recursion model and describe how the programmer needs to reason about this model. Impromptu supports temporal recursion through a single function - the `schedule` call. `Schedule` is an asynchronous call that is responsible for registering an *event tuple* with the scheduling engine. In its simplest form `schedule` takes a deadline time and a closure as its only two arguments. If the closure requires arguments then this variable number of arguments must also be supplied to the `schedule` call.

```
;; perodic cycle called every 1000 ticks
;; with incrementing integer counter
(define periodic
   (lambda (time count)
      (print 'count:> count)
      (schedule (+ time 1000) periodic
                (+ time 1000) (+ count 1))))

;; start cycle
(periodic (now) 0)
```

Listing 1. Temporal Recursion

[2] Actually the garbage collector, which is a concurrent variant of Baker's treadmill [3], runs on its own kernel thread. Therefore each Impromptu process actually requires two kernel threads.

Listing 1 shows the definition of a temporal recursion. It is worth noting that the temporal recursion semantics are structured around the notion of scheduled function execution as opposed to the more abstract concept of a scheduled *event*. We believe that this provides a very minimal extension to existing Lisp programming practice (in particular, the tail-recursive programming style familiar to Scheme programmers). In listing 1 a single temporal recursion is started when `(periodic (now) 0)` is called. As its final action the `perodic` function schedules its own execution in 1000 clock-ticks from `time` and increments both `time` and `count`. The `count` and `time` arguments are the only observable state and are isolated to this particular temporal recursion. A second, or third etc., concurrent temporal recursion can be started at any time by re-evaluating `periodic`. Note that any state (i.e. `count`) is isolated, being discrete for each temporal recursion.

Temporal recursion can also support shared-state. By scheduling a function closed over `count`, it is possible to share this state between multiple temporal recursions. Listing 2 demonstrates the combination of shared state (*count*) and isolated state (*name*) between each temporal recursion. The log output of listing 2 will look like this "A0,B1,C2,A3,B4,C5,A6...".

```
;; shared memory "count"
(define shared-count
  (letrec ((count 0)
           (proc (lambda (time name)
                   (print name count)
                   (set! count (+ count 1))
                   (schedule (+ time *second*) proc
                             (+ time *second*) name))))
    proc))

;; start three temporal recursions which share "count"
(let ((time (now)))
  (shared-count time "A")
  (shared-count time "B")
  (shared-count time "C"))
```

Listing 2. Temporal Recursions sharing memory

Impromptu's temporal recursion is a co-operative concurrency model and the programmer is responsible for ensuring that temporal recursions meet their real-time deadlines. Functions must either run to completion or explicitly yield control. In the simplest case of a single temporal recursion this requirement forces the programmer to ensure that a queued function is capable of completing its execution before the time of its future invocation. This situation becomes more complex once a second temporal recursion is introduced and the programmer must ensure that each of two temporal recursions must not only meet their own deadlines, but ensure that they do not interfere with each others deadlines. This potentially presents a very difficult problem for programmers to reason about but, in practice, the problem is less problematic as the common pattern of use for temporal recursion is to create functions with extremely short execution times. We argue that this convention would not only be characteristic of real-time multimedia, but would be the case with any type of successful, real-time, reactive programming – but it is an issue that the programmer needs to be aware of and a consequence of making Impromptu's concurrency model explicit.

4.3 Time

Impromptu has been designed to provide a reactive system with timing accuracy and precision based on the constraints of human perception. Human auditory perception has a significantly higher precision than visual perception and requires accuracy in the microsecond range[40]. Although this is significantly longer than the nanosecond clock of most current operating systems, the accuracy required to maintain a suitable quality-of-service for audiovisual tasks remains a challenge[25]. WTP makes this challenge even more difficult by demanding that the real-time system be dynamic. It is not only the dynamic number and type of reactive events that can effect real-time performance but also dynamic changes to the execution of the reactive system itself.

The real-time demands of WTP systems are relaxed by recognising that, firstly, human agency combines perception with cycles of cognition and action with combined timings measured in fractions of seconds and, secondly, that timing constraints in the task domain may also be greater than mere perception. For example, in the multimedia performance domain an audience would forgive, or not even perceive, the loss of a single musical note, however the loss of every fifth note might cause some consternation. Thus the quality-of-service demands on the system are "firm" rather than being "hard" real time.

Impromptu includes a firm real-time scheduling engine based on an earliest-deadline-first (EDF) approach[8]. The scheduler supports two clock types, an audio-rate clock, measured in audio samples (or "ticks") since Impromptu was initialised, and a real-world "wall clock", represented as an NTP timestamp[33] [3].

Earlier, we claimed that a WTP programming environment should support a "first class" semantics for time in order to help programmers reason about the temporal state of a program. Lee et al.[26] proposed six features that should be present in order to provide a programming environment with a "first class" semantics for time:

[3] Why does Impromptu have two clocks? An audio-rate clock makes sense for two reasons. Firstly as one of Impromptu's primary task domains is temporally bound to human audio perception a clock referencing the audio-rate is useful and appropriate. Secondly, professional audio devices are usually built with higher quality quartz crystals than commodity computing hardware. However, there are two primary problems with an audio-rate clock. Firstly, an audio-rate clock has no notion of real-world time, making it unsuitable for distributed timing tasks. Secondly, it is often convenient for users to operate in terms of wall-clock time - "please start this video at 3:00pm". The scheduling engine does not discriminate between these two time references and programmers are free to use either and to convert freely between the two.

- The ability to express timing constraints
- Timed communication
- Enforcement of timing constraints
- Tolerance to violations and constraints
- Maintaining consistency in distributed real-time systems
- Static timing verification

We present these six features in the following subsections and outline how Impromptu performs in meeting each requirement.

4.3.1 Ability to express timing constraints

Impromptu provides the ability to express both start-time and execution-time constraints in either clock-time or sample-time. Start-time provides the earliest deadline by which a function must begin executing. Execution-time expresses the maximum time available for the execution of the given function.

```
;; video player temporal recursion looping
;; at a rate of 1/24th of one second.
;; maximum-execution time of 1/32nd of one second
(define video-player
   (lambda (time mov position)
      (let ((frame (gfx:get-movie-frame mov position)))
         (gfx:draw-image time *canvas* frame 1)
         (schedule (cons (+ (now) (/ *second* 24))
                         (/ *second* 32))
                   video-player
                     (+ time (/ *second* 24))
                     mov (+ position 1/24)))))

;; this call starts the temporal recursion
;; the execution deadline for this first call
;; is the default execution duration of the process
(video-player (now) (gfx:load-movie "/tmp/myfilm.mp4") 0.0)
```

Listing 3. MPEG Video Playback

Listing 3 demonstrates a straight forward MPEG video player written in Impromptu. The video-player plays back video at a rate of 24 frames per second. It has a temporal constraint on its execution-time mandating that each frame complete its rendering in less than 1/32nd of a second.

4.3.2 Timed Communication

Impromptu's primary communication mechanism is via remote-procedure calls. Impromptu's RPCs can be either synchronous or asynchronous. Synchronous RPCs are designed to work within the context of Impromptu's temporal recursion model. A synchronous RPC is responsible for capturing a continuation before sending an asynchronous message to the remote process. It then immediately breaks to the top-level. This effectively stalls any active temporal recursion. The result of executing the procedure call on the remote process is returned to the sending process, at which point the sending process executes the stored continuation with the returned result. A time-out can be applied to the RPC in which case any stored continuation will lapse after the time-out period has expired. The time-out is also used as the maximum-execution duration for the remote procedure call.

Asynchronous calls can also specify a time-out although this value is for the remote maximum-execution duration value only as asynchronous RPC calls do not directly return values.

Impromptu's internal NTP support includes clock synchronisation that has been designed to fine-tune the local NTP host-time across a LAN. This additional level of clock synchronisation provides temporal accuracy in the microsecond range for communication between distributed processes [48]. As previously discussed the microsecond range is precise enough to support the synchronisation of audiovisual activities.

4.3.3 System enforcement of timing constraints

Impromptu supports deadline enforcement through both the start-time and execution-time of functions. The Impromptu EDF scheduler is responsible for ensuring that events are dispatched to their requested process, and each process is then responsible for ensuring that each events execution meets its execution-time deadline. Events that the EDF fails to dispatch on time are subject to a global reaper. Events which fail to execute within their stated execution-time deadline raise an exception or call an optionally supplied closure/continuation.

One problem with Impromptu's execution duration time-out is the time spent in foreign function calls. Impromptu does not have the ability to abort computation until the completion of a foreign function call. This can effect the systems ability to respond in a timely manner to calls which exceed their maximum execution duration. However, this excess will be immediately reported on completion of the foreign function call, and the computational will abort.

4.3.4 Tolerance to violations and constraints

Impromptu's temporal constraint violations currently fall into one of two categories. Either an event is late in starting execution, or it is late in completing execution. In the first case there is some limited programmer discretion in setting a reaper time-out.

The scheduler incorporates a global reaper which is responsible for culling events whose start times are late by an amount greater than a user-defined default value (usually in the order of 1ms). Culled events register an error message to the users log view, but are otherwise ignored. There is no similar tolerance for missed execution completion times.

4.3.5 Maintaing consistency in distributed real-time systems

Impromptu provides co-ordination of distributed processes by incorporating a Linda style Tuple Space model[48] known as "Spaces". *Spaces* provides Impromptu with inter-process co-ordination both locally and remotely. It is implemented on top of Impromptu's RPC mechanism and sup-

ports the same time-out semantics. *Spaces* implementation is based on the original Linda model [17] and supports the primitives post, read and remove - or, in Impromptu, write, read and take. Support for regular expression matches and numeric conditionals are supported as tuple parameter matches. Parameter matches can be bound in-place as per the original Linda model.

```
;; no timeout - wait indefinitely for new messages to arrive
;; and then repeat
(define mail-reader
   (lambda (username)
      (let loop ((mailbox (append "mailbox/" username)))
         (spaces:take mailbox 'subject 'from 'message)
         (printf "from:%s\nsubject:%s\n%s"
                 from subject message)
         (loop mailbox))))

;; read andrew's mailbox
(mail-reader "andrew")
```

Listing 4. Linda Co-ordination on Model

4.3.6 Static timing verification

Given the nature of WTP, it is difficult to see how a formal temporal verification of Impromptu programs could proceed. There appear to be two significant impediments to providing such an analysis. Firstly, as an environment designed for WTP, Impromptu programs are extremely dynamic. Not only is it impossible to say with certainty how many, and what type of events the system may be required to process, but it also impossible to know exactly how those events are to be processed before runtime.

Secondly, the system is distributed and highly reliant on integration with external libraries. The culmination of these issues, makes it extremely unlikely that any reasonably-broad static analysis is possible. Instead, Impromptu attempts to compensate by providing programmers with runtime feedback about the temporal state of the executing system and provides programmers with the ability to control overdue tasks through the use of programmer-supplied timing constraints.

4.4 AudioVisual Subsystems

Impromptu provides a substantial runtime system with a particular focus on audio and visual subsystems. Here we very briefly present features of these two major subsystems.

4.4.1 Audio Architecture

Impromptu's audio subsystem implements the Apple AudioUnit specification, a set of design requirements for implementing AudioUnit plugins and the protocols used to connect these nodes into arbitrary data-flow graphs. Additionally, the specification outlines various protocols for communicating between applications which host these DSP graphs and the individual AudioUnit plugins which they instantiate and communicate with. The specification also outlines various user interface guidelines[4] and various conventions for supporting interoperability. The AudioUnit specification is a well-established industry standard supported by the majority of high-end digital audio companies.

The AudioUnit/CoreAudio specification prescribes a synchronous, data-flow system, with an adjustable block size and a pull architecture. In its standard form the AudioUnit specification does not explicitly support multi-threaded operation. However, Impromptu supports the parallel processing of upstream subgraphs.

Impromptu also allows on-the-fly compilation of "custom" AudioUnits at runtime. These "custom" AudioUnits can be integrated anywhere into the DSP graph. This provides Impromptu programmers with the ability to mix and match commercial audio plugins with their own on-the-fly custom DSP code. Listing 5 shows the code used to compile and load a simple low pass filter. This code can be hot-swapped at anytime without adversely effecting the audio signal chain. Listing 5 shows a "custom" low-pass filter.

```
;; DSP kernel for low pass filter
(define lp-filter
   (lambda (sample time channel data)
      (let ((frequency (f64g data 0))
            (p0 (f64g data 1))
            (p1 (f64g data 2)))
        (set! p0 (+ p0 (* frequency (- sample p0))))
        (set! p1 (+ p1 (* frequency (- p0 p1))))
        (f64s data 1 p0)
        (f64s data 2 p1)
        p1)))

(define filter-dat (objc:data:make (* 8 3)))
(f64s filter-dat 0 0.5)
(au:code:load au-code-node "filter" lp-filter filter-dat)
```

Listing 5. Simple LowPass Filter

4.4.2 Graphics Architecture

Impromptu provides access to the OSX graphics system through a custom canvas class. Canvases can be constructed with any dimension and can optionally be run fullscreen on any available visual display. Any number of canvases can be instantiated for either Quartz vector/bitmap drawing or OpenGL rendering.

```
;; create canvas 640x480
(define canvas (gfx:make-canvas 640 480))
;; start live video camera
(gfx:start-live-video)
;; create a gaussian blur image filter
(define *blur* (gfx:make-filter "CIGaussianBlur"))

;; playback live camera images at 24 frames per second
(define draw-camera
  (lambda (time)
    (let* ((image (gfx:get-live-frame))
           (filtered-image (gfx:apply-filter *blur* image)))
      (gfx:draw-image time canvas filtered-image 1)
      (callback (+ time (/ *samplerate* 24)) draw-camera
                (+ time (/ *samplerate* 24))))))

;; start drawing to canvas
(draw-camera (now))
```

Listing 6. Apply gaussian filter to live image stream and render to canvas

[4] AudioUnit plugins may provide their own custom GUI.

The Impromptu graphics subsystem supports OpenGL, GLSL, vector drawing, Bitmap drawing, video decoding and encoding and image processing.

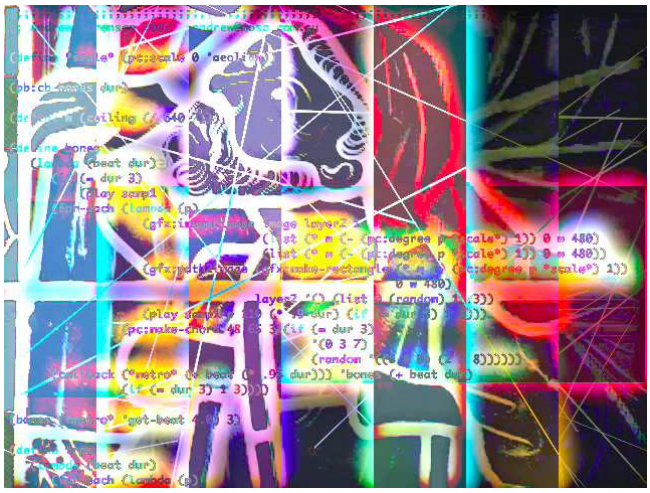

Figure 3. A snapshot of an Impromptu graphics canvas

Graphics animation is handled using Impromptu's temporal recursion paradigm for concurrency. Listing 6 illustrates the application of a gaussian filter to a live video stream at 24 frames per second.

A second temporal recursion can be run at a completely separate frame-rate, or indeed at a variable frame-rate, with both rendering to a single, or multiple canvases. Audiovisual synchronisation is trivial as the same temporal structures are used for both audio and visual change-over-time.

4.5 Related Work

There are far too many areas of related work to cover in this small section. Instead, we will briefly outline a few major influences and related projects.

Languages such as Lisp [29] and Smalltalk [23] have influenced Impromptu's interactive nature. Self [44], Boxer [1], SubText [15] and SuperGlue [31] have explored the idea of direct manipulation of the process domain through the program domain. Simula [13], Erlang [2] and Linda [17] have all influenced Impromptu's concurrency model.

A precursor to this paper's central theme of WTP comes from Rohrhuber et al.[41] in their presentation of JITLib, a programming extension for the SuperCollider language (and where Just In Time is used in the same way that we refer to WTP). They discuss WTP in relation to real-time audio signal processing and examine some of the concerns raised by dynamic, temporal programming.

The first explicit reference to a real-time, temporal recursion, appears to come from the music programming language Moxie [9]. Moxie defines a `cause` operation which can be used to schedule temporal recursions. Dannenburg's CMU MIDI Toolkit borrowed `cause` from Moxie [14]. *Engines* [20, 21] and *coroutines* [13, 34] both support similar concurrency styles to temporal recursion although they are commonly used in implicit abstractions and are generally not implemented with real-time interrupt scheduling.

There is a long history of research into real-time temporal semantics in the embedded systems community. Ada, RT-Java [8] and the synchronous languages Lustre, Esterel and Signal [5] are a few of the many languages for embedded programming which support a temporal semantics. However, these languages have different design goals to WTP environments. They are primarily designed for product development, rather than experimentation, and as a result they commonly lack the dynamic and interactive nature required for WTP. They are often designed for demanding real-time applications, where static analysis is applied to tightly specified behaviour in order to provide hard real-time temporal guarantees.

There has been a substantial body of research into Functional Reactive Programming (FRP) as a paradigm for modelling continuous *behaviours* and discrete *events* [12, 16, 35, 52]. In many respects FRP is similar to synchronous systems, including ChucK which is discussed below. FRP shares many of the same advantages as the synchronous languages (such as formal reasoning) and some of the same disadvantages (such as performance and interfacing to non-synchronous systems). FatherTime (FrTime) has attempted to address some of these concerns by implementing FRP using asynchronous and impure methods [11]. It is implemented in Scheme and has had some limited exposure to livecoding practice through the Fluxus [18] project.

SuperCollider [30] and ChucK [53] are two of the few programming environments that have directly addressed the notion of WTP. Both environments are heavily oriented towards audio signal-processing and algorithmic music composition and are commonly used for livecoding. Both environments share many of the same motivations as Impromptu although they are less focused on visual applications.

SuperCollider written by James McCartney shares many architectural features with Impromptu. It is a dynamic, bytecode interpreted language with a deterministic co-operative concurrency model, real-time garbage collection and real-time EDF scheduling. SuperCollider's concurrency model is asynchronous.

ChucK, developed by Ge Wang and Perry Cook follows in the synchronous languages genre (Signal, Lustre, Esterel [5]) in implementing a concurrency model with temporal semantics based around the synchrony hypothesis. ChucK's data-flow [22] "shreds", are explicit, user space and deterministic.

ChucK's synchronous approach provides one of the major distinctions between the environments. ChucK attempts to provide a similar (synchronous) semantics for programming tasks at both low-level *continuous* and higher level *discrete* temporal levels [5]. This provides for an elegant, unified semantics for programmers. However, a purely synchronous

[5] Continuous in this context meaning changing every tick of some clock

implementation strategy makes distributed and integration programming difficult as synchronous code does not naturally co-ordinate with non-synchronous external libraries and non-deterministic distributed computation. SuperCollider and Impromptu's choice of asynchronous concurrency trades ChucK's clean semantics for a more flexible architecture.

```
// define runner function
~runner = { |msg, k, inc|
      { |time| // lambda closing over msg k inc
           // print integer message
           (msg ++ k).postln;
           // increment k
           k = k + inc;
           // loop in 1 second
           SystemClock.schedAbs(time+1.0,thisFunction);
      }.value(thisThread.seconds); // call anonymous lambda
};

// start first temporal recursion
~runner.value("TR-1",0,1);
// start second temporal recursion
~runner.value("TR-2",0,2);
```

Listing 7. Spawning two SuperCollider Timed Loops

An example of this dual semantics in Impromptu is the code in Listing 5 which includes on-the-fly compilation of DSP code in Impromptu. That example is *implicit* with regards to both time and concurrency because it is compiled into a node in a synchronous data-flow graph. It is dynamic, but is semantically removed from the generally explicit and asynchronous temporal recursion model used for other Impromptu programming [6].

```
// define shred function
fun void shred_runner (string msg, int k, int inc)
{
    while(true)
    {
        // print integer with message
        <<< msg, k >>>;
        // increment integer
        k + inc => k;
        // loop in 1 second
        1.0::second => now;
    }
}

// start first shred
spork ~ shred_runner("shred-1",0,1);
// start second shred
spork ~ shred_runner("shred-2",0,2);
```

Listing 8. Spawning two ChucK shreds

Code Listings 7, 8 and 9 compare similar behaviour in SuperCollider, ChucK and Impromptu. All three support an explicit temporal semantics. They all share concurrency models that are deterministic and lightweight supporting hundreds or thousands of concurrent activities. Neither SuperCollider or ChucK directly support multi-core processing, although they are capable of communicating between multiple separate instances of their runtimes. Both SuperCollider and ChucK provide strong start-time constraints although neither environment currently includes support for execution-duration constraints.

[6] It is worth mentioning that Impromptu's temporal recursion framework could be used for signal processing (i.e. low-level *continuous* processing), but this would fit less comfortably with the AudioUnit specification and is less efficient.

```
;; define temporal recursion function
(define runner
   (lambda (time msg k inc)
      ;; print integer message
      (print msg k)
      ;; loop in 1 second and increment integer
      (schedule (+ time *second*) runner
                (+ time *second*) msg (+ k inc) inc)))

;;start first TR
(runner (now) "tr-1" 0 1)
;; start second TR
(runner (now) "tr-2" 0 2))
```

Listing 9. Spawning two Impromptu "temporal recursions"

5. Discussion

5.1 Cyber-physical systems

Drawing temporal and concurrency concerns into the program domain brings the activity of programming into the domain of cyber-physical systems where programmers become agents with a cyber-physical relationship with the world. As computing becomes increasingly dominated by cyber-physical systems, the traditional, data-transformational view of computing passed down from Church and Turing [25] [7] becomes increasingly incommensurate with computational demands[43]. As expressed by Edward Lee:

> ... why is the latency of audio signals in modern PCs a large fraction of a second? Audio processes are quite slow by physical standards, and a large fraction of a second is an enormous amount of time. To achieve good audio performance in a computer (e.g. in a set-top box, which is required to have good audio performance), engineers are forced to discard many of the innovations of the last 30 years of computing. They often work without an operating system, without virtual memory, without high-level programming languages, without memory management, and without reusable component libraries, which do not expose temporal properties on their interfaces. Those innovations are built on a key premise: that time is irrelevant to correctness; it is at most a measure of quality. Faster is better, if you are willing to pay the price. By contrast, what these systems need is not faster computing, but physical actions taken at the right time. It needs to be a semantic property, not a quality factor. [25]

It is not that computer science has not attempted to reconcile a physical time with a computational time but, rather, the level of abstraction at which this reconciliation usually occurs. Modern operating systems, routinely provide high-precision clocks and operate successfully in highly concurrent and media-rich environments. Nevertheless, temporality is lost when moving to higher-level languages and runtime

environments because real-world time is not of primary concern to the computational process for many programming tasks.

With Time Programming is a paradigm which meets many of Lee's key concerns. With Time Programmers are active participants in the world and have a fundamental interest in time and concurrency. Their role, as human agents in a cyber-physical relationship with the world, is not only reactive but also intentional. They become an integral part of a cyber-physical system by providing stimulus for the system and reacting responsively to its perturbations in the world. Significantly, their role is not merely gestural (or direct) but also procedural, and procedural control provides the ability to plan for future states and to adapt to changing requirements computationally. Providing WTP programmers with the ability to act procedurally in a reactive feedback loop with the physical world requires suitable time and concurrency semantics. In particular, we see the need for a "first-class" semantics for time and a deterministic concurrency paradigm. For WTP, one of the primary ramifications of such first-class semantics for time is the possibility of aligning development time, computational time and real-world clock-time.

5.2 Conclusion

We have discussed a programming paradigm which we have called With Time Programming and we have claimed that WTP is an extension of just-in-time programming where the programmer's real-world causal agency is supported by real-time programming infrastructure. WTP is directed towards experimental and experiential practices where an environmental artefact is the primary goal.

We have described a WTP programming environment, Impromptu, and we have discussed how Impromptu's support for accurate start-time and execution-time constraints, in combination with a simple, lightweight concurrency pattern, helps a programmer to reason about, to control and to react within a multimedia cyber-physical system. In the future, we are interested in applying Impromptu to other domains, such as robotics and command-and-control, in situations where human agency is of central concern.

We are entering an age in which many computing activities will no longer be primarily algebraic and transformational. Instead, computing devices will increasingly interact with each other and the world around them in cyber-physical systems. The work described here is part of a continuing exploration into the act of programming as a real-time causal activity where the human programmer is an active agent in a cyber-physical world.

References

[1] A. A diSessa and H. Abelson. Boxer: a reconstructible computational medium. *Communications of the ACM*, 29(9):859–868, 1986.

[2] J. Armstrong. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[3] H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27:66–70, 1992.

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development, 2001. *Online: http://www.agilemanifesto.org*, 2001.

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[6] R. Beraldi and L. Nigro. Distributed simulation of timed petri nets. *IEEE CONCURRENCY*, 7(4):52–62, 1999.

[7] W. Beynon, R. Boyatt, and S. Russ. Rethinking programming. In *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*, pages 149–154, 2006.

[8] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison Wesley, 2001.

[9] D. Collinge. Moxie: A language for computer music performance. pages 217–220. International Computer Music Conference, ICMC, 1984.

[10] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2004.

[11] G. Cooper and S. Krishnamurthi. Frtime: Functional reactive programming in plt scheme. *Computer science technical report. Brown University. CS-03-20*, 2004.

[12] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, page 18. ACM, 2003.

[13] O. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):678, 1966.

[14] R. Dannenberg. The cmu midi toolkit. In *Proceedings of the 1986 International Computer Music Conference*, pages 53–56, 1986.

[15] J. Edwards. Subtext: Uncovering the simplicity of programming. In *In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 505–518. ACM, 2005.

[16] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273. ACM, 1997.

[17] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[18] D. Griffiths. Fluxus, 2010. URL `http://www.pawfal.org/fluxus/`.

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

[20] C. Haynes and D. Friedman. Abstracting timed preemption with engines* 1. *Computer Languages*, 12(2):109–121, 1987.

[21] R. Hieb and R. Dybvig. Continuations and concurrency. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 128–136. ACM, 1990.

[22] R. Karp and R. Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, pages 1390–1411, 1966.

[23] A. Kay. Smalltalk, a communication medium for children of all ages. *Xerox Palo Alto Research Center, Palo Alto, CA*, 1974.

[24] E. Lee. Concurrent models of computation for embedded software. *System-on-chip: next generation electronics*, page 223, 2006.

[25] E. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009.

[26] I. Lee, S. Davidson, and V. Wolfe. Motivating time as a first class entity. *Technical Reports (CIS)*, page 288, 1987.

[27] T. Magnusson. Ixilang. URL `http://www.ixi-audio.net/ixilang/`.

[28] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: a sneak preview [education]. In *Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on*, pages 104–109, 2004.

[29] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):195, 1960.

[30] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[31] S. McDirmid. Living it up with a live programming language. In *Proceedings of the 2007 OOPSLA conference*, volume 42, pages 623–638. ACM New York, NY, USA, 2007.

[32] A. McLean. Hacking perl in nightclubs. *at http://www. perl. com/pub/a/2004/08/31/livecode. html*, 2004.

[33] D. Mills. Rfc 1305-network time protocol (version 3) specification. *Implementation and Analysis*, 1992.

[34] A. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):6, 2009.

[35] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, page 64. ACM, 2002.

[36] S. Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books New York, 1980.

[37] R. Potter. Just-in-time programming. *Watch What I Do: Programming by Demonstration, MIT Press, Cambridge, MA*, pages 513–526, 1993.

[38] M. Puckette. Max at seventeen. *Computer Music Journal*, 26 (4):31–43, 2002.

[39] R Project for Statistical Computing. `http://www.r-project.org/`.

[40] C. Roads. *Microsound*. The MIT Press, 2004.

[41] J. Rohrhuber, A. de Campo, and R. Wieser. Algorithms today notes on language design for just in time programming. In *International Computer Music Conference*, page 291. ICMA, 2005.

[42] E. Sandewall. Programming in an interactive environment: the"lisp"experience. *ACM Computing Surveys (CSUR)*, 10(1): 35–71, 1978.

[43] B. C. Smith. *On the origin of objects / Brian Cantwell Smith*. MIT Press, Cambridge, Mass. :, 1996. ISBN 0262193639 0262692090.

[44] R. Smith and D. Ungar. Programming as an experience: The inspiration for self. *Object-Oriented Programming*, pages 303–330.

[45] A. Sorensen. Impromptu: an interactive programming environment for composition and performance. *Proceedings of the Australasian Computer Music Conference*, 2005.

[46] A. Sorensen. Oscillating rhythms, 2008. URL `http://www.acid.net.au`

[47] A. Sorensen. Livecoding screencasts, 2010. URL `http://vimeo.com/impromptu/videos/sort:plays`.

[48] A. Sorensen. A distributed memory for networked livecoding performance. In *International Computer Music Conference*. ICMA, ICMA, June 2010.

[49] A. Sorensen and A. Brown. aa-cell in practice: An approach to musical live coding'. In *Proceedings of the International Computer Music Conference*, pages 292–299, 2007.

[50] D. Souflis and J. Shapiro. Tinyscheme. URL `http://tinyscheme.sourceforge.net`.

[51] TOPLAP. Toplap website. URL `http://www.toplap.org`.

[52] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252. ACM, 2000.

[53] G. Wang, P. Cook, et al. Chuck: A concurrent, on-the-fly audio programming language. In *Proceedings of International Computer Music Conference*, pages 219–226. Citeseer, 2003.