Impromptu: A few notes on implementation

Andrew Sorensen

November 23, 2009

1 Impromptu Language Implementation

Impromptu has been developed along evolutionary and pragmatic lines and strives to:

- Be highly reactive and distributed with minimal latency
- Be dynamic and runtime configurable
- Provide an exploratory, open environment in a high level language
- Support efficient low-level runtime programability
- Provide efficient audio and video processing engines
- Provide a semantics for time.
- Be predicable and reliable
- Be fault tolerant and provide useable runtime feedback
- Be expressive and fun to use

The following sections outline Impromptu's various systems and the ways in which it succeeds of fails in addressing these points.

1.1 Impromptu Processes

In order to take advantage of modern hardware, particularly multi-core CPU's, Impromptu supports a notion of "process". Processes in Impromptu are preemptive Mach kernel threads managed by the OSX operating system. Each process maintains its own Scheme heap memory and Scheme execution stack. New Impromptu processes can be spawned at runtime via a call to `ipc:new-process`. Processes in Impromptu are initialised and identified by a unique name (text string).

Communication between Impromptu processes is via message-passing with no direct access to another processes heap memory. Inter-process messaging occurs via TCP, transparently bridging local or remote processes. Each process provides a FIFO queue which acts as a mailbox for incoming messages. Messages are sent asynchronously. Messages are read serially from the mailbox in FIFO order. Inter-process messages can contain any first class serialisable Scheme data, including lambda expressions (i.e. code). Impromptu provides an inter-process communications (IPC) API which abstracts much of this detail from users. Only three IPC calls are used in regular production work `ipc:define`, `ipc:call` and `ipc:new-process`. An additional asynchronous call is available `ipc:call-async` for cases where the sending process does not wish to block waiting for a result returned from the receiving process.

Each process uses its own *mostly* parallel garbage collector which also runs as an independent OSX Mach kernel thread. The Impromptu real-time parallel GC is based on Baker's four colour treadmill [Baker, 1992]. The GC runs the collector on a separate Mach kernel thread with mutator access protected by a write barrier. The GC is *mostly* parallel in the sense that object collection is done in parallel. The collection thread is blocked while the colour bit is inverted and root nodes are marked.

Currently the GC does not include generational support, which would probably increase the performance of the GC. In particular, adding generational support would help with faster deallocation of large short term Objective-C memory allocations which occur on the OSX process heap - not on the Impromptu process heap.

There has been no formal verification of the Impromptu parallel garbage collector aside from its regular (and reliable) use in the wild.

The impromptu Scheme interpreter is based on the TinyScheme interpreter [Souflis and Shapiro], although it has been substantially altered since

its initial integration into the project in 2005.

1.2 Scheduling Tasks

Impromptu includes an asynchronous firm[1] real-time scheduling engine based on an earliest deadline first approach[Burns and Wellings, 2001]. The scheduler is *naive* and makes no attempt to enforce deadlines, with all tasks running to completion. Aside from the tasks deadline there is no priority distinction made between tasks. Task deadlines are represented as as absolute time in audio samples since impromptu's initialisation. Audio samples per second being driven by the clock of an audio device. Typically this resolution will be 44.1k ,48k ,96k or 192k samples[2] (ticks) per second depending on the type of audio device and its current setting. To maximize system performance, the scheduling engine is quantised to the block size of the audio signal processing graph. This block size is user definable but defaults to 64 samples. An audio device running at 48k would therefore result in 48000/64 calls to the scheduling engine per second. The maximum resolution of a task in this instance would be approximately 1.3 milliseconds. An audio device running at 192k with 16 sample block sizes would refine this resolution down to less than 100 microseconds. Although large in the scheme of computer hardware architectures these times are within the bounds of aural and visual perception making them acceptable for most audiovisual scheduling[3]. Additionally Impromptu also provides the ability to compile synchronous data flow code that is run at the sampling rate of the current device. This DSP compilation is discussed in more detail later.

Of greater significance to audio and visual production is a requirement that the clock and scheduling engine do not drift over time. Furthermore, the impromptu architecture allows audio tasks - which require a much smaller temporal resolution than visual tasks - to include an offset into the sample block, ensuring that any audio affect is introduced into the signal pro-

[1]Where a firm real-time system is distinct from soft real-time because late events, while not catastrophically effecting the system, do catastrophically effect the task domain. Music makes a good example. A real-time music system may continue to function properly even after the delivery of a late musical event. However, the musical performance will be detrimentally effected by this single late event.

[2]Independent of number of audio channels or audio busses

[3]Indeed this is far greater resolution than is required by most of the real-time tasks that we usually consider

cessing graph sample accurately. Impromptu, also supports the notion of a real-world clock via calls to `clock`. Real-world clock time can be translated into impromptu's sample base and vice versa `clock->samples` and `samples->clock`.

Impromptu supports two base notions of task execution. Tasks which call C++ methods and tasks which call Scheme closures, procedures and continuations. Generally speaking C++ tasks are used to execute temporally sensitive media related calls such as playing a note, drawing to a graphics context etc., or communications related tasks such as sending OSC or MIDI messages. Scheme code tasks maintain a closure, procedure or continuation until called upon to execute the given task in a given impromptu process. Any required arguments are maintained along with a closure or procedure while continuations maintain their own stack. As an example we can schedule a `print` function to append a message to a system log in one minute from now:

```
(callback (+ (now) *minute*) print "helloworld")
```

This example demonstrates the `callback` function; which accepts a time *(+ (now) *minute*)*, procedure *print*, and single argument *"hello world"*. `callback` takes responsibility for passing the procedure, argument and target process to the scheduling engine. At the appointed time the scheduler will send this task back to the requested process which maybe running on the local machine or on a remote host.

Scheme tasks are buffered on a given processes FIFO queue before being evaluated, in turn, by that process. Buffering messages through a FIFO queue allows the scheduler to continue processing other tasks, passing other Scheme tasks off to other interpreters or executing C++ directly. C++ tasks are executed immediately, without being buffered, and do not require scheme evaluation. Thus, c++ tasks are more efficient and are therefor used for tasks requiring high precision.

In practice the system is surprisingly efficient, providing perceptibly accurate feedback to temporally sensitive musical/visual tasks. In part this is because media rendering tasks, such as playing a note or drawing to the screen, are ultimately performed by C++ tasks which are executed directly by the scheduler with no network or process buffer latency. In order not to overload the scheduler Impromptu minimises these C++ tasks to extremely time critical events such as sample accurate audio tasks.

A single central scheduling engine is shared by all Impromptu processes.

This allows processes to synchronise accurately to a single shared clock. Impromptu programmers access this shared clock via calls to `(now)` which returns the time - in audio samples - since Impromptu was initialised. Uses working on disparate hosts can use the `clock` calls to provide NTP based system clock times as the basis for communication.

1.3 Concurrency

Impromptu offers two distinct concurrency models, a preemptive threaded model suitable for multicore processing, and a corporative multitasking method for "in process" concurrency. The preemptive threaded model is implemented through Impromptu's process model. In this model there is no shared memory with all communication using message passing. The "in process" cooperative multitasking model utilises the asynchronous scheduling engine to support aperiodic tasks. Impromptu uses a pattern of usage dubbed "temporal recursion" to model lower level mechanisms such as green threads. This is in direct contrast to most real-time languages that provide lower level constructs which users can then use to program aperiodic tasks [Burns and Wellings, 2001].

The asynchronous cooperative model is the primary concurrency model for implementing musical and graphical temporality and is cooperative in nature. The notion is simple. As the last executable action a given code block schedules itself for execution at some future time. This recursive and asynchronous process provides a simple, and I would argue intuitive, model for concurrency with the primary caveat being that developers must take active responsibility for managing/conceptualising time.

```
(define player
   (lambda (pitch duration)
      (play-note (now) piano pitch 65 duration)
      (if (< pitch 72)
          (callback (+ (now) duration) player
                    (+ pitch 1)
                    (random (list 5000 10000))))))

(player 60 5000)
```

Listing 1: Temporal Recursion

The code listing above implements a *temporal recursion* that plays a chromatic scale beginning on middle C (60) and ending on the C one octave above (72). The notes are played serially with the first duration being 5000 samples long and each subsequent duration being stochastically chosen (5000

or 10000). The closure `player` schedules itself as its final action (providing that pitch is less than 72) by calling `callback`. It passes a time to call itself back at `(+ (now) duration)`; `player` (itself) as the function to call back into and two additional arguments required by `player`'s function signature - `[pitch (+ pitch 1)] and [duration (random (list 5000 10000))]`.

By calling `player` multiple times, multiple temporal recursions can be run concurrently, resulting in a polyphonic (multi instrument), or concurrent activity. Temporal recursions provide data encapsulation through argument passing, when working with closures, or by scheduling continuations which maintain a copy of their own stack.

Impromptu can schedule closures, or continuations, a primary distinction being whether state is passed explicitly or implicitly. Consider these two pieces of code, whose resulting execution is identical but who's implementations are slightly different.

```
;; play 10 notes (recursive async style)
(define async
  (lambda (cnt)
     (play-note (now) piano 60 80 10000)
     (if (< cnt 9)
         (callback (+ (now) 10000) async (+ cnt 1)))))

(async 0)

;; play 10 notes (iterative sync style)
(define sync
  (lambda ()
     (dotimes (i 10)
        (play-note (now) piano 60 80 10000)
        (sys:sleep 10000))))

(sync)
```

Listing 2: Synchronous and asynchronous implementation styles

The first of these two functions, `async`, passes state explicitly through `cnt` which is updated and stored by `callback` each time `async` is rescheduled. Alternatively `sync` uses `sys:sleep` to implicitly capture a continuation and pass the continuation to the scheduling engine. This is important conceptually because these two styles emphasise either a synchronous or an asynchronous view of time. In reality this distinction is more fundementally based on whether state is implicitly or explicity managed. But to the impromptu user sys:sleep appears to *hold up time* in a synchronous manner. To show that behind the scenes these two calls (`sys:sleep` and `callback`) are both asynchronous `sys:sleep` is trivially implemented on top of `callback`.

```
(define *sys:toplevel-continuation* '())
(call/cc (lambda (k) (set! *sys:toplevel-continuation* k)))

(define sys:sleep
   (lambda (duration)
      (call/cc (lambda (cont)
                 (callback (+ (now) duration) cont #t)
                 (*sys:toplevel-continuation* 0)
                 #t)))))
```

Listing 3: Implementing sleep with callback

One important consequence of Impromptu's choice of concurrency models is that the programmer is freed from the direct consideration of critical sections. Although this freedom comes with a price, being the requirement that the programmer must explicitly manage time. Race conditions are still possible but by removing temporal indeterminism it is much simpler for programmers to reason about state. Indeed programmers must go out of there way to introduce race conditions.

1.4 Compilation with LLVM

Impromptu supports runtime JIT compilation of scheme code by providing a front-end to the LLVM compiler architecture [Lattner and Adve, 2004]. Impromptu compiles scheme to LLVM's intermediate representation and then uses the LLVM backend to compile to x86 machine code. Impromptu supports this functionality through the `(sys:compile <closure>)` function, which accepts a single scheme closure and returns a scheme foreign function (x86). The result can be bound and used as a normal scheme procedure call. This dynamic functionality allows scheme code to be compiled on-the-fly. The LLVM JIT runtime gives high level programmers access to low-level features and performance.

```
;; opengl drawing function
(define draw-gl
   (lambda (angle gl)
      (gl:clear gl (+ *gl:depth-buffer-bit* *gl:color-buffer-bit*))
      (gl:load-identity gl)
      (gl:translate *gl* 0 0 -3.0)
      (dotimes (i 50.0)
         (gl:rotate gl angle (/ i 50.0) 1.0 1.0)
         (glut:wire-cube gl (tan (/ i 50.0))))))

;; opengl drawing loop
(define draw-loop
   (lambda (angle)
      (gl:lock-context *gl*)
      (draw-gl angle *gl*)
      (gl:unlock-context *gl*)
      (gl:flush *gl*)
      (callback (now) 'draw-loop (modulo (+ angle .25) 360))))
```

Listing 4: OpenGL drawing

The simple example above outlines a temporal recursion `draw-loop` which calls some opengl drawing code `draw-gl`, looping as fast as possible. In this state the system is interpreting the scheme closure `draw-gl` on each loop. The `sys:compile` call will compile this closure on-the-fly and if the returned compiled foreign function `draw-loop` is bound to the draw-gl symbol the draw-loop closure will seamlessly switch to calling the foreign function code. For example `(define draw-gl (sys:compile draw-gl))`. The intermediate representation after compiling the draw-gl scheme closure is shown below.

```
define void @draw-gl(double %angle,double %gl)
{
entry:
  %val577 = add i64 256, 16384
  %cast580 = trunc i64 %val577 to i32
   call void @glClear(i32 %cast580)
   call void @glLoadIdentity()
  %cast583 = sitofp i64 0 to double
  %cast584 = sitofp i64 0 to double
   call void @glTranslated(double %cast583,double %cast584,double -3.0)
   br label %loop586

loop586:
  %i = phi double [ 0.0, %entry],[ %next586,%closeloop586]

  %val591 = fdiv double %i, 50.0
   call void @glRotated(double %angle,double %val591,double 1.0,double 1.0)
  %val599 = fdiv double %i, 50.0
  %res596 = call double @tan(double %val599)
   call void @glutWireCube(double %res596)
  %cast602 = sitofp i64 1 to double

  %next586 = fadd double %i, %cast602
  %cmp586 = fcmp ult double %i,50.0
   br label %closeloop586

closeloop586:
  br i1 %cmp586, label %loop586, label %after586

after586:
  ret void
}
```

Listing 5: OpenGL drawing LLVM IR

The above example illustrates the importance of being able to hot swap compiled and interpreted code. In order to make this feel natural to the programmer the impromptu compiler tries to remove the need for duel-semantics by supporting basic type inferencing as well as some on-the-fly semantic comversions. At present this type inference system is quite limited (no recursive types for example) but is an area of ongoing development.

The LLVM compiler is a new addition to the Impromptu system, providing low-level systems programming support to the high level Scheme environment. The compiler is a work in progress but has already made a significant contribution to the Impromptu system by affording efficient run-time compilation of OpenGL and audio DSP code. It is anticipated that the continued development of the impromptu compiler will be a major ongoing area of work.

2 Audiovisual Subsystems

Beyond the language runtime, Impromptu is composed of two major subsystems. It is these subsystems that are at the heart of the Impromptu runtime, providing the task domain infrastructures of audio and vision that the Impromptu programmer is actively engaged in manipulating at runtime. These subsystems leverage large sections of the OSX developer frameworks - Cocoa, CoreAudio, CoreMIDI, CoreGraphics, OpenGL, Quicktime and CoreImage.

Access to these subsystems is generally provided through one of three mechanisms; C++ methods that can be asynchronously called from the Impromptu scheduling engine; C foreign functions which are called synchronously from scheme and Objective-C methods that can be called synchronously from the scheme interpreter through the Impromptu Objective-C bridge. Of course abstractions can be made such that these facilities interoperate, so in effect the primary decision to be made for any given piece of functionality is whether synchronous or asynchronous access is required. Most commonly access is synchronous.

2.1 Sound

2.1.1 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. Additionally the specification 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, although some AudioUnit's[5] support multi-threaded con-

[4] AudioUnit plugin's may provide their own custom GUI

[5] NativeInstruments Kontakt for example

currency within their own execution context. Additionally multiple HAL's (Hardware Abstraction Layers) can operate in parallel providing the ability to run distributed audio processing across cores/processors.

Impromptu interfaces directly to the CoreAudio/AudioUnit API giving programmers the ability to directly manipulate AudioUnit's and AudioUnit graph configurations programatically at runtime. The following list provides an overview of the types of runtime programmable control available for interacting with AudioUnits:

- instantiate, connect and re-arrange AudioUnit nodes at runtime.
- display, and allow interaction with, custom and generic AudioUnit GUI's.
- send MIDI and extended note data to AudioUnit's with sample accurate offsets
- read and write parameter and property data with sample accurate offsets
- display information about the state and properties of an AudioUnit
- display information about the state/configuration of the AudioUnit graph.

Impromptu also ships with a range of AudioUnit's specifically tailored for use in a real-time programatic context. Lower level objects such as oscillators, filters, noise generators, sample-players, recorders, analysers and alike provide Impromptu programmers with access to lower-level signal processing tools.

2.2 Low-level DSP Code

Impromptu also supports the ability to compile and hot-swap audio DSP code on-the-fly. Impromptu ships with a "Custom Code" AudioUnit whose DSP kernel can be swapped in at runtime. By default the "Custom Code" AudioUnit passes on any incoming signal unaltered, however this kernel can be swapped out at runtime and replaced with x86 machine code compiled with Impromptu's compiler. Any number of "Custom Code" AudioUnits can be instantiated and added to the signal graph as either generators that do

not take an input, or effects that accept an incoming signal. "Custom Code" AudioUnit's can optionally share binary data with the Impromptu scheme runtime allowing scheme code and AudioUnit code to interact efficiently. This functionality provides Impromptu with the ability to build and modify efficient DSP code at runtime. Below is an example of a simple low-pass filter kernel. The DSP kernel expects to receive: an incoming sample, a time in samples for the incoming sample, a channel for the incoming sample and binary data object which is shared with the scheme runtime.

```
;; DSP kernel
(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)))

;; allocate 24 bytes of shared heap memory
(define filter-dat (objc:data:make (* 8 3)))
;; set default frequency to 0.5 in first 8 bytes of shared heap
(f64s filter-dat 0 0.5)
;; compile and load lp-filter kernel into au-code-node
(au:code:load au-code-node "filter" lp-filter filter-dat)
```

Listing 6: Simple LowPass Filter

This DSP kernel can then be loaded into a particular "Custom Code" AudioUnit node by calling `au:code:load` which is responsible for compiling the kernel and hot-swapping the compiled kernel into the supplied AudioUnit node. This change can be made at anytime and the "Custom Code" AudioUnit ensures that the new code is swapped in-between audio frames.

2.2.1 Sound Communications

Impromptu provides a number of mechanisms for communicating either musical, audio or control information. Aside from the TCP mechanisms available for IPC communication Impromptu supports the OSC communication protocol, a general purpose communications specification for communication over UDP Wright [2002]. OSC has become an industry standard in recent years and looks set to replace the venerable MIDI protocol as the primary industry messaging protocol. Many audiovisual applications support the OSC protocol making it a good choice for cross apllication/network communication.

The OSC protocol is very flexible and Impromptu implements most of the specification including timetags, bundles and arrays. There are a number of initiatives investigating various timesyncing protocols to sit on top of OSC. Impromptu has implemented a version of Linda's tuple spaces for remote synchronization by providing a shared, distributed, memory Carriero and Gelernter [1989].

Impromptu supports MIDI both as an inter-host communication protocol for sending messages to AudioUnit plugins as well as providing MIDI reading and writing to and from external devices, via virtual ports (other applications running on the same OSX host), or to and from files.

Audio data can be streamed in real-time via one of two systems built into Impromptu. Firstly, audio-data can be streamed via Apple's *NetSend* and *NetReceive* AudioUnit's. These two AudioUnit's provide end points for a streamed network session and can be placed anywhere in Impromptu's signal processing chain. *NetSend* and *NetRecieve* offer various formats for streaming to support a wide range of network bandwidths.

Additionally Impromptu has built-in support for iChat Theatre which allows audio and visual data to be streamed through any iChat session. If started in an iChat session Impromptu will automatically start streaming audio and any Impromptu graphics canvas (either OpenGL or Bitmap), can be used as the video source for the streaming session. This facility potentially allows for remote performances as well as presentations and alike.

Impromptu has a number of options for streaming audio to and from disk.

2.3 Graphics

Impromptu provides access to the OSX graphics system through a custom canvas class that supports threadsafe drawing[6]. 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.

The Impromptu graphics subsystem supports OpenGL, GLSL, Quartz vector drawing (NSBezierPaths the like), Bitmap drawing (NSBitmapImageRep and the like), video decoding and encoding (any Quicktime format) and image processing (CIFilter CIImage). Additionally, Impromptu provides

[6]allowing multiple Impromptu processes to draw concurrently to any canvas

access to Apple's vDSP, vImage and veclib libraries providing a host of vector optimised image/data processing functions.

Graphics animation is handled using the standard Impromptu temporal recursion paradigm. Animation becomes a trivial task. The example below illustrates a temporal recursion applying a gaussian filter to live video frames at 24 frames per second.

```
;; 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 ()
      (let* ((camera-image (gfx:get-live-frame))
             (filtered-image (gfx:apply-filter blur camera-image)))
         (gfx:draw-image (now) canvas filtered-image 1)
         (objc:release (+ (now) 5000) camera-image filtered-image)
         (callback (+ (now) (/ *samplerate* 24)) draw-camera))))

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

Listing 7: Apply gaussian filter to live image stream and render to canvas

A second termporal recursion (concurrent activity) can be run at a completely separate frame-rate, or indeed at a variable frame-rate, with both rendering to a single canvas or any number of separate canvases. Audiovisual synchronisation is trivial as the same temporal structures are used for both audio and video change over time. In this example the `objc:release` call is used to rapidly release memory consumed by the camera-image and filter-image, specifying a future time for their release. As previously discussed, adding generations to the GC should address this problem removing the need for an explicit call to `objc:release`.

Impromptu provides access to one or more OpenGL contexts and supports a substantial subset of the OpenGL API including support for the OpenGL shading language. There are a number of mechanisms for sharing data between OpenGL and Quartz - a Quartz canvas can used an an OpenGL texture for example. OpenGL calls can be compiled for significantly faster execution.

Any Quartz or OpenGL canvas can be recorded to a Quicktime movie along with any extant audio output.

3 Extensibility

As a practical solution for real-world media software development it is important that Impromptu provide a relatively open architecture for third party extension. Impromptu provides a range of extension options already touched upon, including audio plugins and custom dsp code, image processing kernels, and the OpenGL shading language. More generally Impromptu provides extension through a bi-directional bridge to the Objective-C runtime.

Impromptu supports Objective-C objects natively within its dynamic type system. ObjC objects are automatically managed by the GC being retained and released automatically. ObjC class and instance methods can be called and new objects can be instantiated[7].

The code below shows a simple application that instantiates a Cocoa Window with a single slider whose changing values are printed to the Impromptu log.

```
(define window (objc:make "NSWindow"
                          "initWithContentRect:styleMask:backing:defer:"
                          (list 200 200 400 300) 1 2 0))

(objc:call window "orderFront:" 0)
(define contentView (objc:call window "contentView"))
(define slider (objc:make "NSSlider" "initWithFrame:" (list 20 20 30 200)))
(objc:call slider "setTarget:" *objc:bridge*)
(objc:call slider "setAction:" "floatAction:")
(objc:call slider "setTag:" 1)
(objc:call contentView "addSubview:" slider)

(define objc:action
  (lambda (id val)
    (print val)))
```

Listing 8: ObjC-bridge calls for creating an OSX Window with Slider

The Objective-C bridge enables programmers to write custom OSX applications without needing to leave the Impromptu runtime. An Objective-C bridge API is provided to further extend integration by allowing programmers to build custom Objective-C frameworks which can also call back into an Impromptu process. Native Cocoa GUI's can be built inside Impromptu or can be loaded as Interface Builder NIB or XIB files. Interface components (NSViews etc.) can communicate directly with Impromptu through the bridge. This supports the rapid development of user interfaces in real-time.[8].

[7]At this time the Objective-C bridge does not allow classes to be extended at runtime

[8]Examples of "standalone" OSX applications written entirely within Impromptu can

The Objective-C runtime provides access to an extensive set of libraries supporting networking, data persistence, web protocols and rendering engines, string parsing, file I/O, serialisation, XML parsing etc..

References

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

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

N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):458, 1989.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04*, 2004.

D. Souflis and J.S. Shapiro. Tinyscheme. URL `http://tinyscheme.sourceforge.net`.

Matthew Wright. Open sound control 1.0 specification. 2002. URL `http://opensoundcontrol.org/spec-1_0`.

be downloaded here