COMMON MUSIC 3

Heinrich Taube
University of Illinois
School of Music

ABSTRACT
Common Music [1] Version 3 (CM3) is a new, completely redesigned version of the Common Music composition system implemented in C++ and Scheme and intended for interactive, real-time composition. The system is delivered as a cross-platform C++ GUI application containing a threaded scheme interpreter, a real-time music scheduler, graphical components (editor, plotter, menu/dialog control), and threaded connections to audio and midi services. Two different Scheme implementations can be used as CM3’s extension language: Chicken Scheme [2] and SndLib/S7 [3], by William Schottstaedt. When built with SndLib/S7 CM3 provides a fully integrated environment for algorithmic composition and sound synthesis delivered as a relocatable (drag-and-drop) application that runs identically on Mac OSX, Windows Vista and Linux.

1. INTRODUCTION
Common Music (CM) is a music composition environment that produces sound by transforming a high-level, representation of music into a variety of control protocols for sound synthesis and display. Earlier versions of CM were implemented in Common Lisp, which allowed for a highly portable and expressive system but without real-time scheduling or a portable graphical user interface. The new major version (3) branch of Common Music has been completely redesigned for interactive composition and GUI tools. It is implemented in C++ and Scheme to provide an efficient yet expressive platform for algorithmic composition. The C++ side of CM provides a portable, drag-and-drop application framework built using the JUCE [4] cross-platform toolkit. The scheme side provides dynamic programming functionality. The Scheme and C++ sides are tightly integrated; scheme code typically calls C++ to do much of its work and C++ calls back into Scheme whenever users evaluate Lisp code or receive asynchronous data from audio/midi devices. From the user’s perspective the two languages are seamlessly integrated: Lisp code is edited in C++ GUI windows and Scheme execution uses C++ windows to display its results. As a C++/Scheme hybrid CM3 shares characteristics with both Snd and Impromptu [5] but differs in a number of important respects: CM3 is cross platform, drag and drop; it supports both real-time and file based composition; and it is designed to work with multiple types of audio targets: midi/audio ports, syntheses languages (Sndlib and Csound), even music notation applications using FOMUS [6] and MusicXML.

2. APPLICATION DESIGN AND DELIVERY
The CM3 source tree builds both a GUI and a non-GUI version of the Common Music runtime. The GUI version is intended to be used as a stand-alone environment for algorithmic composition. The non-GUI version can be used that can be used in toolchains These applications share an identical library of core services but differ in how the services are delivered. Core services include a real-time event scheduler, a threaded scheme interpreter, threaded audio and midi ports, access to host OS services, and Common Music's algorithmic composition toolbox. All run time code, examples, tutorials and documentation are contained in a zip (compressed) archive embedded in the application that is automatically accessed by components that use it. This means that the application is fully relocatable (drag-and-drop): the only data that is saved to the disk outside the application is a settings file that preserves the user’s environment across multiple executions of the program. The application automatically caches settings whenever the user changes important program state such as selecting a midi input device or specifying a new default audio sampling rate.

2.1. Grace
The Grace application (Graphical Real-time Algorithmic Composition Environment) is a completely graphical program for algorithmic composition (Figure 1). Grace contains a multi-purpose (main) Console window, a point-and-click code editor with Emacs emulation, Lisp and SAL [7] evaluation services, syntax highlighting and symbol documentation look-up, a general purpose Plotter for graphing, editing and sequencing data, and a menu and dialog interface for configuring and working with the various audio and midi targets provide by the application.
2.2. cm

The cm application is a lightweight executable that delivers Common Music's core services as a console (text) application suitable for running inside a Terminal or as an inferior process inside Emacs. If used inside Emacs an optional sal.el file can be loaded that defines a SAL Emacs mode for editing and evaluating SAL programs (a similar Scheme mode is built into Emacs.) Since the cm application has no GUI components all audio and midi configuring is accomplished by executing Scheme or SAL expressions that call into the same underlying core library that the GUI components use.

![Figure 1. Some graphical components in the Grace application: the Console (output) window displays the CM/Scheme interpreter in the upper left, the audio instrument browser is upper right, code editors with Scheme syntax highlighting are at the bottom and an audio file player is floating on top.](image)

3. SCHEDULING

Common Music’s built-in scheme interpreter runs in its own thread that is forked as soon the application starts running. All communication between the C++ side and the Scheme interpreter is managed by the Scheme thread’s run() loop executing on a 1ms tick. The run() loop maintains a priority queue of time sorted ‘scheme nodes’ that it checks each time through its loop. When the queue is empty the thread simply blocks until a node is available, otherwise each node is processed when its time stamp is no greater than the current time. The scheduler currently utilizes three types of nodes: eval, hook, and process nodes. Eval nodes contain Scheme or SAL text to be read as expressions and evaluated by Scheme. Eval nodes are routinely added by GUI components such the Editor whenever the user presses Command-Enter to execute an expression. These nodes normally have a ‘0’ time stamp to insure that they are added to the front of the queue and executed as soon as possible. Hook nodes are nodes that contain data arriving asynchronously from audio/midi ports or from GUI components in other process. The user registers Scheme/SAL hook functions with the scheduler to receive this data. Hook functions can do whatever they want with the data they receive, for example save it in a recording buffer, or use it to trigger real time musical processes. This demonstrates the tight connection between Scheme and C++ in Common Music 3: callbacks from C++ can trigger Scheme code that sends data back to C++ which then causes additional Scheme callbacks. This happens, for example, when a composer ‘sprouts’ a musical algorithm, called a process, to run in real time.

3.1. Music Processes

The third type of scheduling node is called a process node. A process is a Scheme function that is added to the scheduler at a specific (real) time using CM’s ‘sprout’ function (Figure 2). When the sprouted node’s time is current the scheduler executes a callback into Scheme using the node’s function pointer; the function then executes as a normal Scheme function. Any Scheme code can be executed in this manner but a typical process would likely send data at the current (real) time to an audio port that is open on the C++ side. Each time the scheduler evaluates the process node it tests the function’s return value to determine what to do next. If the function returns boolean false then the node is not rescheduled (and the process ‘dies’), otherwise if the return value is a positive number then the process is added back into the scheduling queue to be re-evaluated again that many seconds in the future.

3.2. Score Scheduling (Faster Than Real-Time)

The scheduler normally runs in real-time mode, which means that nodes wait in the queue until their sprout time is the current real time. However, if a musical process is sprouted with an output file (.mid, .sco, .wav) then the scheduler will execute the processes synchronously, as fast as possible, to write the output file ‘atomically’ under the sprout call; the resulting file can then be post-processed as soon as the sprout call returns (Figure 2). Thus Common Music 3 supports both interactive and file-based composition styles with the same scheduling model.

```scheme
(define process simp(len, lb, ub)
  run repeat len
    send "mp:midi", key: between(lb, ub)
    wait pick(.1, .2, .4)
  end
  sprout list(simp(10, 60, 90), simp(10, 30, 60))
)```
sprout simp(100, 20, 90), "~/test.mid"

Figure 2. A musical process definition (SAL) and scheduling. The first sprout runs two copies of the simp process in real-time, the second sprout runs faster than real time to generate a midifile containing simp’s output.

4. THE COMMON MUSIC 3 TOOLBOX

The Common Music 3 algorithmic composition toolbox is implemented as a collection of C++ and Scheme functions, variables and record types. Many of the Scheme functions call into CM’s underlying C++ support library to do their work. Offloading portions of the computation from Scheme to C++ speeds up execution for real-time work and also eliminates the generation of Lisp ‘garbage’ during the calculation of intermediate (non-returned) values.

The demands of real-time scheduling also triggered major design changes to the calling conventions of the operators defined in the toolkit: as a rule the CM3 API is leaner, more consistent and more powerful than the comparable API in the CM2 branch. The distinguishing features of Common Music 3 operators compared to the CM2 (Common Lisp) version are:

1. The CM3 tool kit does not require an object system or use generic functions. It is written in R5RS compliant Scheme [8] with the addition of records, as provided by SRFI-75 [9].

2. SAL and Scheme equivalence. Algorithms and live coding in CM3 can be programmed in the ‘native’ Scheme or in SAL, an easy-to-learn infix language specifically designed for algorithmic music composition. Editors in Grace support both languages with syntax highlighting, Emacs-like cursor motion over expressions and evaluation services. SAL’s lexer is written in C++ and its parser is implemented in Scheme to permit future releases to support users adding their own commands to the language at runtime.

3. A terse, clean function calling syntax minimizes typing to help assist ‘interactive coding’: function and variables have short names and never require the use of quoted symbols or Lisp keywords in argument data.

4. Many functions in the API are overloaded to automatically process Lisp lists as well as atomic data. Consistent, built in list mapping reduces the number of functions in API and minimizes the need for writing ad hoc loops to process Lisp data during interactive (live) programming.

5. All functions that accept variable arguments also permit a single list argument to be passed, eliminating the need to call apply explicitly (Figure 3).

print plus(1,2,3)
6
print plus([1 2 3])
6
print plus([1 2 3],4)
{5 6 7}

Figure 3. Function overloading automates Lisp list processing and reduces the necessity of explicit looping.

(5) Optional function parameters (if any) can be provided positionally and/or using DSSSL style keyword parameters (Figure 4).

(6) The ‘send’ macro provides a single, consistent interface for routing data to audio targets such as midi ports, audio files and Csound scores. Every audio target implements a set of methods to accept and process data sent to it by the user. The first argument to the send macro is its message, a literal string in the format “target:method” that identifies a specific audio resource and method to receive the data sent to it.

send "mp:midi", 0, 1, 60, .5
send "mp:midi", 0, key: between(60, 90)

Figure 4. The send macro routes data to audio targets, in this example data is being routed to the Midi Port’s midi method. The second example demonstrates shows that the macro supports both positional and keywords styles of value passing.

Note that send messages are implemented by lower-level Scheme functions that can be called directly to achieve the same effect. Using send has two benefits: (1) it makes it very clear that the code is manipulating audio data and (2) the send macro performs argument syntax checks at read time (rather than run time) to help prevent errors that would otherwise only be caught during the real-time execution.

5. AUDIO TARGETS

Audio targets are sources and destinations for audio and MIDI data. CM depends on the cross-platform, low latency Audio/Midi classes provided by the JUCE tool kit to do most of this work. Each audio target provided by CM implements its own unique set of send methods to accept formatted data sent to it by the user.

5.1. Midi Port

Common Music’s Midi Port target provides direct connection to external MIDI devices and the computer’s
MIDI application bus. Connections can be opened or closed by evaluating code or by using the Audio menu.

5.1.1. MidiOut

The Midi Output port is threaded so that a composer can send ‘future’ messages directly to the port; these messages ‘wait’ at the port until their time stamp becomes current. The MidiOut port also has built-in support for microtuning. Composers can select between 16 quantizing levels, from the (default) semi-tonal quantization down to 1/16 of a semitone (roughly 6 cents). When the port is placed in a micro-tonal resolution any floating point key number data kk.kc that arrives at the port is interpreted as cc cents above the MIDI key number kk and quantized to the port’s current micro-tonal resolution.

5.1.2. MidiIn

The MidiIn port captures messages arriving from external midi devices and application on the host’s MIDI bus. Messages that arrive can be filtered by type and channel so that only specific messages are caught. Messages can be saved in a recording buffer and/or passed asynchronously to a hook function that the user specifies in Scheme. Midi input hook are passed three arguments: the midi opcode of the arriving event and up to two data bytes of message values.

5.1.3. Midifile

The Midi port’s internal buffer allows messages to be imported and exported to midi files.

5.2. SNDLIB

Common Music’s SndLib target connects to the SndLib audio library developed by William Schottstaedt at CCRAM. If Common Music is built with SndLib/S7 then the full CLM audio distribution, including its complete set of audio instruments, is built into the Scheme interpreter. The SndLib library supports sample-by-sample audio synthesis and audio file processing. SndLib instruments are specialized scheme functions that compute audio using SndLib’s suite of unit generators and optimized run loop. CLM instruments can be loaded into Scheme at any point. A specialized instrument browser in Grace provides the complete CLM instrument distribution in a sortable table, by using this browser all database of instruments can be loaded along with examples of their usage (Figure 1).

5.3. Csound

Common Music’s Csound target provides score file generation and (optional) audio post-processing of Csound score files using an installed Csound application. The target provides a configuration dialog in Grace to include the location of the Csound executable, optional command invocation arguments, orchestra files and playback options.

6. CONCLUSION

Common Music 3 is a major new version of the Common Music compositions system. It replaces Common Lisp with a portable C++ application framework and an embedded Scheme interpreter as its extension language. The source tree builds two different applications, a fully-relocatable, drag-and-drop GUI environment called Grace and a console application called cm. Both applications provide a threaded scheme interpreter and threaded connection to audio/midi targets. The scheme interpreter runs inside a real-time scheduling thread that manages input and output sent to/from scheme from the surrounding C++ application. The result is a flexible, real-time composition environment that leverages the strengths of both languages and delivers, from the user’s perspective, a seamlessly integrated environment for audio processing and algorithmic composition.

7. REFERENCES


