Csound6: old code renewed

John FITCH and **Victor LAZZARINI** and **Steven YI**
Department of Music
National University of Ireland
Maynooth,
Ireland,
{jpff@codemist.co.uk, victor.lazzarini@nuim.ie, stevenyi@gmail.com}

Abstract

This paper describes the current status of the development of a new major version of Csound. We begin by introducing the software and its historical significance. We then detail the important aspects of Csound 5 and the motivation for version 6. Following, this we discuss the changes to the software that have already been implemented. The final section explores the expected developments prior to the first release and the planned additions that will be coming on stream in later updates of the system.

Keywords

Music Programming Languages, Sound Synthesis, Audio Signal Processing

1 Introduction

In March 2012, a decision was taken to move the development of Csound from version 5 to a new major version, 6. This meant that most of the major changes and improvements to the software would cease to be made in Csound 5, and while new versions would be released, these will consist mainly of bug fixes and minor changes (possibly including new opcodes). Moving to a new version allowed developers to rethink key aspects of the system, without the requirement of keeping ABI or API compatibility with earlier iterations. The only restriction, which is a fundamental one for Csound, is to provide backwards language compatibility, ensuring that music composed with the software will continue to be preserved.

This paper describes the motivation for the changes, current state of development and prospective plans for the system.

1.1 Short History of Csound

1.1.1 Early History

Csound has had a long history of development, which can be traced back to Barry Vercoe's MUSIC 360[Vercoe, 1973] package for computer music, which was itself a variant of Max Mathews' and Joan Miller's MUSIC IV[Mathews and Miller, 1964]. Following the introduction of the PDP-11 minicomputer, a modified version of the software appeared as MUSIC 11[Vercoe, 1981]. Later, with the availability of C (and UNIX), this program was re-written in that language as Csound[Boulanger, 2000], allowing a simpler cycle of development and portability, in comparison to its predecessor.

The system, in its first released version, embodied a largely successful attempt at providing a cross-platform program for sound synthesis and signal processing. Csound was then adopted by a large development community in the mid 90s, after being translated into the ANSI C standard by John ffitch in the early half of the decade. In the early 2000s, the final releases of version 4 attempted to retrofit an application programming interface (API), so that the system could be used as a library.

1.1.2 Csound 5

The need for the further development of the Csound API, as well as other innovations, prompted a code freeze and a complete overhaul of the system into version 5[ffitch, 2005]. Much of this development included updating 1970s programming practices by applying more modern standards. One of the major aims was to make the code reentrant, so that its use as a library could be made more robust. In 2006, version 5.00 was released. The developments embodied by this and subsequent releases allowed a varied use of the software, with a number of third-party projects benefitting from them.

1.2 Csound operation in a nutshell

As a MUSIC-N language, Csound incorporates a compiler for instruments. During performance, these can be activated (instantiated) by various means, the traditional one being the standard numeric score. In Csound 5, compilation can only be done once per performance run, so new instruments cannot be added to an already running engine (for this performance

needs to be interrupted so the compilation can take place).

The steps involved in the compiler can be divided into two: parsing, and compilation proper. The first creates an abstract syntax tree (AST) representing the instruments. The compilation then creates data structures in memory that correspond to the AST. When an instrument is instantiated, an init-pass loop is performed, executing all the once-off operations for that instance. This is then inserted in a list of active instruments, and its performance code is executed sequentially, processing vectors (audio signals), scalars (control signals) or frames of spectral data. The list orders instruments by ascending number, so higher-order ones will always be executed last. All of the key aspects of Csound operation are exposed by the API.

2 Motivation

In the six years since its release, Csound 5 continued to develop in many ways, mostly in response to user needs, as well as providing further processing capabilities in the form of new opcodes. After a long gestation, early in 2012, the new flex-bison parser was completed and added as a standard option. This was the final major step of development for Csound, where the last big chunk of 1970s code, the old ad-hoc parser, was replaced by a modern, maintainable, and extendable parser. Following the 2011 Csound Conference in Hannover, it was clear that there were a number of user requests that would be more easily achievable with a rethink of the system. Such suggestions included:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- facilities for live coding

The time was ripe for major changes to be made. User suggestions prompted developers to begin an internal cleanup of code, the removal of older components (such as the old parser), and a reorganisation of the API. It was also an opportunity to code-walk, and with that find inconsistencies and bugs that would normally be hidden. In particular, changes related to repeated loading and compilation of new instruments would require (and indeed force) a welcome separation of language and synthesis engine, which is well underway at present.

3 Developments to date

3.1 Build System and Tests

In Csound 5, the official build system is SCons[1]. Over time, a CMake-based[2] build was introduced and used for local developer use, as well as later for Debian packaging and iOS builds. In Csound 6, the official build system is now the CMake-based build. Moving to CMake introduced some hurdles and changes in workflow, but it also brought with it generation of build system files, such as Makefiles, XCode projects, and Eclipse projects. This solved a problem of IDE-based projects for building Csound becoming out of sync with changes in the SConstruct file for SCons, as well as brought more ways for developers to approach building and working with Csound code, particularly through IDE's.

Using the CTest feature in CMake, unit and functional tests have been added to Csound 6's codebase. CTest is the test running utility used to execute the individual C-code tests. In addition, CUnit[3] is employed to create the individual tests and test-suites within the test code files. In addition to C-code testing, the suite of CSD's used for application/integration testing continues to grow, and a new set of Python tests has also been added for testing API usage from a host language.

3.2 Code reorganisation

The Csound code base is passing through a significant reorganisation. Firstly, parts of it that are now obsolete, such as the old parser, have been removed. Some opcodes with special licensing conditions that have been deemed not to be conducive to further development have been completely rewritten (also with some efficiency and generality improvements). The CSOUND struct has been rationalised and reorganised, with many modifications due to the various changes outlined in the next sections.

[1] http://www.scons.org
[2] http://www.cmake.org
[3] http://cunit.sourceforge.net

Finally, the public API is going through a redesign process (details of which are discussed below).

3.3 Type system

The Csound Orchestra language uses strongly typed variables and enforces these at compile-time. This type information is used to determine the size of memory to allocate for a variable as well as for specifying the in- and out-arg types for opcodes. The system of types used prior to Csound 6 was hard-coded into the parser and compiler. Adding new types would require adding code in many places.

In Csound 6, a generic type system was implemented as well as tracking of variable names to types. The new system provides a mechanism to create and handle types, such that new types can be easily added to the language. The system also helps clarify how types are used during compilation. Another feature is that variable definitions and types were previously discarded after compile-time; in Csound 6, this information is kept after compilation. This allows the possibility of inspecting variables found in instruments or in the global memory space.

3.4 Generic Arrays

In Csound 5, a 't' type was added that provided a user-definable length, single-dimension array of floating-point numbers. In Csound 6, with the introduction of the generic type system, the code for t-types was extended to allow creation of homogenous, multi-dimensional arrays of any type. Additionally, the argument list specification for opcodes was extended to allow denoting arrays as arguments.

3.5 On-the-fly Compilation

The steps necessary for the replacement or addition of new instruments or UDOs to a running Csound engine, or, more concisely, on-the-fly compilation, started to be taken in the latter versions of Csound 5. It was, of course, sine-qua-non to have a properly structured parser, which we did in 5.17. Also, as a side-effect from the Csound for Android project, compilation from text files was replaced by a new core (memory) file subsystem, so now strings containing Csound code could be presented directly to the parser.

The first step in Csound 6 was made by breaking down the monolithic API call to compile Csound (`csoundCompile()`) into `csoundParseOrc()` and `csoundCompileTree()`, as well as by the addition of a general `csoundStart()` function to get the engine going. The parsing function creates an abstract syntax tree (AST) from a string containing Csound code. The compilation function then creates the internal data structures that the AST represents, ready for engine instantiation(see figure 1).

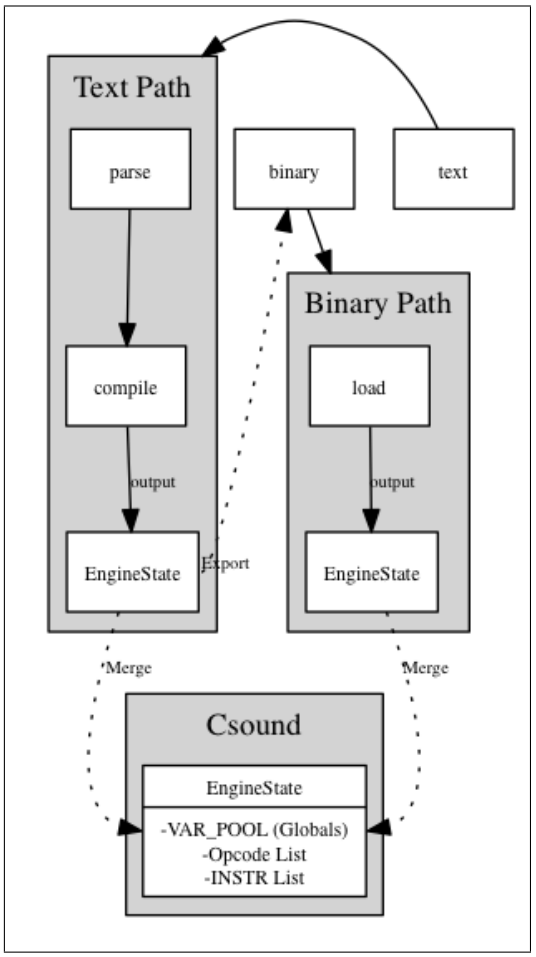

Figure 1: Csound compilation and engineState.

These modifications provided the infrastructure for changes in the code to allow repeated compilation. For this, we have abstracted the data objects relating to instrument definition into an engineState structure. On first compilation, Csound creates its global instrument 0, which is made up of the header statements, global variables declared outside instruments and their initialisation. It then proceeds to compile any other instruments defined in the orchestra (including UDOs, which are a special kind of instrument). On any subsequent compilations, instruments other than 0 are added to a newly-created engineState. After compilation, the new engineState is merged into the current one be-

longing to the running Csound object.

Instrument definitions with the same name or number will replace previously existing ones, but any instances of the old definitions that are active are not touched. New instances will use the new definition, and replaced instruments get added to a deadpool for future memory recovery (which will happen once all old instances are deallocated). A similar process applies to UDOs.

Currently, no built-in thread-safety mechanisms have been placed in the API, so hosts are left to make sure compilation calls are not made concurrently to audio processing calls. However, it is envisaged that the final API will provide functions with built-in thread safe as well as ordinary calls.

3.6 Sample-level accuracy

Csound has always allowed sample-level accuracy, a feature present since its MUSIC 11 incarnation. However, a performance penalty was incurred, since the requirement for this was to set the size of the processing block (`ksmps`) to 1 sample. Code can become very inefficient, since there is a single call of an opcode performance function for each sample of output and this is in conflict with caching.

In Csound 6, an alternative sample accuracy method has been introduced. This involves setting an offset into the processing block, which will round the start time of an event to a single sample. Similarly, event durations are also made to be sample accurate, as the last iteration of each processing loop is limited to the correct number of samples (see figure 2). This option is provided with the non-default `--sample-accurate` flag, to preserve backward compatibility.

Tied events[4] are not subject to sample accurate processing as they involve state reuse and are, in its current form, incompatible with the mechanism. Real-time events are also not affect by the process, as event sensing works on a ksmps-to-ksmps basis. Events scheduled to at least one control-cycle ahead can be made to be sample accurate through this mechanism.

The changes needed for this mechanism to work were significant. Each opcode had to be modified to take account of the offset and end position. The scheduler had to be altered so the start of all events was truncated, instead of rounded, to ksmps boundaries, and the calculation of event duration had to be modified. The offset and end position had to be properly defined for each event, as well as set and reset at specific times for each instrument instance.

[4]In Csound, it is possible to have instrument instances that take up a previously-used memory space, which allows the 'tieing' of events, in analogy to slurs in instrumental music

3.7 Realtime priority mode

Csound has been a realtime audio synthesis engine since 1990. However, it was never provided with strict realtime-safe behaviour, even though in practice, it has been used successfully in many realtime applications. Given the multiple applications of Csound, it makes sense to provide separate operation modes for its engine. In Csound 6, we introduce the realtime priority mode, set by the `--realtime` option, which aims to provide better support for realtime safety, with complete asynchronous file access and a separate thread for unit generator initialisation.

3.7.1 Asynchronous file access

For Csound 6, a new lock-free mechanism has been introduced and some key opcodes have been modified to use it when operating in realtime. It uses a circular buffer, employing an interface which had been already present in Csound (used previously only for lock-free realtime audio). It shares the common file IO structure adopted throughout Csound, with a similar, but dedicated interface. For specific file reading/writing requirements, though, as required for instance by diskin, diskin2 or pvsfwrite, the general interface is not suitable. For this case, special opcode-level asynchronous code has been designed.

3.7.2 Unit generator initialisation

Another important modification of the engine in realtime priority mode is the spawning of a separate thread that is responsible for running all of the unit generator initialisation code. This is more commonly known as the 'init-pass', which is separate from synthesis performance ('perf-pass'). In this mode, when an instrument is instantiated, the init-pass code is immediately run in a separate thread. Once this is done, an instrument is allowed to perform. What this does is to prevent any interruption in the synthesis performance due to non-realtime-safe operations in the initialisation code (memory allocation, file opening, etc.). A side-effect of this is that in some situations, an instrument may be prevented to start performing straight away, as

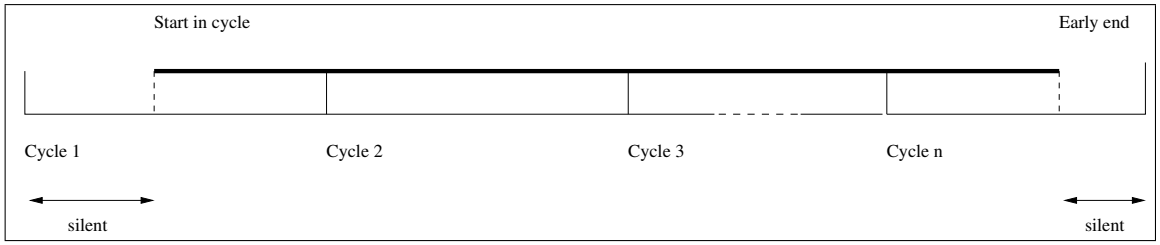

Figure 2: Sample accurate scheme.

the initialisation has not been done. However, this is balanced with the gains in uninterrupted performance.

3.8 Multicore operation

In 2009 an experimental system for using multiple cores for parallel rendering of instruments was written [Wilson, 2009], and this was later incorporated in the standard Csound [ffitch, 2009]. While the design was generally semantically correct it only delivered a performance gains in the case of low control rate and computationally heavy unit generators. Profiling the code showed that the overheads in creating and consuming the directed acyclic graph (DAG) of dependencies, and especially in memory allocation activity.

For Csound 6 we are developing a different approach, that while maintaining the semantic analysis only needs to rebuild the DAG when a score event starts or stops, and in use does not call for changes in the structure. The clue is in the use of watch-lists as found in SAT-solvers [Brown and Purdom Jr, 1982; **?**]. For each task we only need to watch for the completion of one of the dependencies; when a task finishes it can release any task that is waiting for it, and for which all other precursors have already finished. This strategy is also possible with no locking of critical sections, and can use atomic swap primitives instead.

At the same time some simplification of the semantics-gathering has been achieved. This scheme preserves the order-semantics that Csound has always had, but offers efficient utilisation of multiple cores with threads without user intervention beyond saying how many threads to use for the performance stage. Initial measurements (see table 3.8) are very encouraging, in most cases providing significant speed-up. We are continuing to work on possible optimisations.

4 Further work

4.1 Pre-release prospective development (i.e. the "todo list")

The final feature set of Csound 6 is still not finalised. There are a number of possible enhancements that we are considering; some grow from the changes we have described above, and some are long-standing desires.

The introduction of separate compilation and replaceable instruments naturally suggests that we could add a fast loadable format for instruments, building on for example LISP FASL formats, and API and opcode access to loading. It remains to be seen if the source version is sufficiently fast, and whether we can solve the semantic issues that arise, such as f-table independence. What is needed is to document the abstract syntax tree that the parser produces, and thus allow advocates of alternative orchestra languages to provide them.

A restriction in Csound than has long been an irritation is the limit of one string in a score statement. Previous work in this area has attempted to allow up to four strings, but this is both limiting and still buggy. The radical solution would be to introduce a flex/bison parser for the score language and take the opportunity for rethinking the score area. A small start has been made, but the need to support users and the amount of effort needed here has relegated this work to a later release. Until then a simpler scheme will have to be tried for the interim.

The Csound suite of software include a number of analysis programs, most dating from an early time, and written without regard of floating point formats or byte order. From time to time this has caused problems. The task here is to redefine these formats to indicate at least their formats, or even to make the readers capable of format transformations. This needs to be done at some stage and this break seems like a good moment.

With the introduction of on-the-fly compilation one can consider that a user might main-

-j	CloudStrata	Xanadu		Trapped...		
	ksmps=500 (sr=96000)	ksmps=10	ksmps=100	ksmps=10	ksmps=100	ksmps=1000
1	1	1	1	1	1	1
2	0.54	0.57	0.55	0.75	0.79	0.78
3	0.39	0.40	0.40	0.66	0.76	0.73
4	0.32	0.39	0.33	0.61	0.72	0.70

Table 1: *Relative performance with multiple threads in three existing Csound code examples, -j indicates the number of threads used.*

tain a long-running Csound binary and use it for different tasks at different times. This suggests that the current command-line options or API equivalents may need to change at some time after the initialisation. Some changes may be easy, but some may require re-engineering of parts of the engine. We have not yet realised to use-changes that the compilation change will engender.

The new API still needs to be refined. In response to what has been discussed above, we plan, for instance, to expose the configuration parameters in some form (currently held in the OPARMS data structure). At the moment, there is a simple provision for setting separately specific configuration items in the API (as flags). This is to be substituted by a more flexible form, via the exposing of the OPARMS or an OPARMS-like struct to API users.

A number of other changes are planned, some of which are already present in an early form. For instance, the various stages of parsing, compilation, and engine start are now exposed in the provisional API (as detailed for instance in 3.4). There is a plan to provide built-in thread-safety, so some functions can be used directly in a multi-threading environment without further synchronisation or resource protection. The software bus, which now exists in three forms, will be unified to a single mechanism.

4.2 Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation)
- further language features (e.g. namespaces, functions with more than one argument, tuples)
- a system for streaming linear predictive coding processing (in similar fashion to PVOC)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

5 Conclusions

In this paper, we have sought to examine the current development status of Csound 6, as well as the motivations for the fundamental re-engineering of the code that has been underway. We hope to have demonstrated how the technology embodied in this software package has been renovated continuously in response to developments in Computer Science and Music. Our aim is to continue to support a variety of styles of computer music composition and performance, as well as the various ways in which Csound can be used for application development. It is also important to note, for readers, that the re-engineering of Csound is taking place quite publicly in the Csound 6 git repository on Sourceforge (`git://git.code.sf.net/p/csound/csound6-git`). Anyone is welcome to check out and examine our struggles with computer technology and the solutions we are putting forward in this paper.

6 Acknowledgements

Our thanks go to the Csound community for their indulgence, suggestions and support. In addition Martin Brain introduced the idea of watch-lists and co-developed the detailed performance algorithm. We also acknowledge the implicit support from Sourceforge hosting

References

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design.* MIT Press, February.

Cynthia A. Brown and Paul Walton Purdom Jr. 1982. An Empirical Comparison of Backtracking Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(3).

John ffitch. 2005. The Design of Csound5. In *LAC2005*, pages 37–41, Karlsruhe, Germany, April. Zentrum für Kunst und Medientechnologie.

John ffitch. 2009. Parallel Execution of Csound. In *Proceedings of ICMC 2009*, Montreal. ICMA.

M. Mathews and J. E. Miller. 1964. *MUSIC IV Programmer's Manual*. Bell Telephone Labs.

B. Vercoe. 1973. *Reference manual for the MUSIC 360 language for digital sound synthesis*. Studio for Experimental Music, MIT.

B. Vercoe. 1981. *MUSIC 11 Reference Manual*. Studio for Experimental Music, MIT.

Christopher Wilson. 2009. Csound Parallelism. Technical Report CSBU-2009-07, Department of Computer Science, University of Bath.