From the concept of sections to events in Csound

Pedro Kröger[1]

[1]Escola de Música da UFBA
Parque Universitário Edgard Santos, Canela
Salvador, BA, 40150–480

kroeger@pedrokroeger.net

Abstract. *This paper approaches some solutions involving the division of the csound score in smaller parts to reduce its rendering time to a minimum. The ultimate solution involves the use of events. Besides solving elegantly the problem, the definition of events make possible the creation of scores with a hierarquical structure. This concept has been implemented in Monochordum, a compositional environment for Csound.*

1. Introduction

Csound uses two files as input, the orchestra and the score file. Traditionally only one monolithic score file is used per composition. Unfortunately, Csound does not have any device to compile[1] only separated parts so the full composition is rendered even if only one part was modified. Nevertheless, it is not very productive to wait for the rendering of the entire piece only to hear one small section.

In this paper some solutions for this problem will be proposed based in splitting the score into smaller parts and using the utility for recompilation *Make*. The goal is to reduce the time for recompilation to a minimum.

Finally, a solution will be introduced involving the use of events, a structure between the list of notes and the section. The use of events not only solves elegantly the problem but allows the creation of hierarquical structures. Besides the possibility of defining blocks of events, it is possible to define the relationship between these events. This concept has been implemented in Monochordum, a complete compositional environment for Csound.

2. Solutions

2.1. Manual splitting

Commenting sections out is a primitive way to select parts to render. Although this procedure works with small files, it is impracticable with larger ones with hundreds of lines. A better way is to use the Csound's `#include` command. Sections are saved in separated files and called in a main file with the `#include` command (ex. 2.1). The sections not going to be rendered can easly be commented out. For example, section 3 in ex. 2.1 will not be compiled.

This procedure has some advantages; it only uses Csound (not needing any external tool), and arbitrary sections can be selected (e.g., sections 1 and 3).

[1]In this article the word "compile" is used as synonym for "render", i.e., run Csound over an orchestra and score files to get some output (usually a sound file).

Example 2.1 The main score file

```
#include :section-1.sco
s
#include :section-2.sco:
s
;#include :section-3.sco:
e
```

The disadvantages are: the selected sections are always fully compiled, even if nothing was modified at all; there are more files to manage, and the `#include` command has a long history of bugs.

Using Make. The previous solution can be much more automatized with *Make*. The same file structure as in ex. 2.1 is kept, but the main score file is discarded.

The Make tool was created to automatize the process of program compilation. It can recompile only the necessary files based in the modified source files. Although widely used to manage computer programs, "make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change". (Stallman and McGrath 1998, p. 1)

A deeper description of Make is out of the scope of this paper, for more information please refer to the manual.

The ex. 2.2 shows a simple use of Make. The rule `sec1.wav` is defined and can be called by typing `make sec1.wav` in a terminal.

Example 2.2 Rule for a section

```
sec1.wav: sec1.sco
   Csound -Wo sec1.wav Main.orc sec1.sco
```

Since we don't have the main score anymore we need some way to mix the sections together. The Csound *mixer* tool or programs like *ecasound* and *sox* can be used for this purpose.

The real power of Make can be seen in ex. 2.3. The rule `Main.wav` has the rules `sec1.wav`, `sec2.wav`, and `sec3.wav` as dependencies. That means that to be able to perform the command in the "Main.wav" rule the three dependencies have to be complete. If one of them is not, Make will automatically compile **only** the missing section. An overview of the process is shown in fig. 1.

Example 2.3 The mixer

```
Main.wav: sec1.wav sec2.wav sec3.wav
   mixer -T 0 sec1.wav \
   -T 120 sec2.wav -T 160 sec3.wav
```

Conclusion. By splitting the score into separated files and using Make, the renderization of individual sections is possible, giving more flexibility and speed. The second solution shows an advance in comparison with the first, since in the later the renderization time is not as small as in the former. Nevertheless, in both solutions the files have to be created

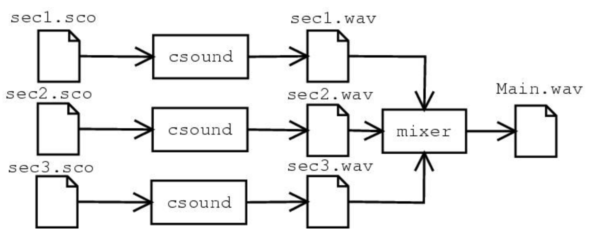


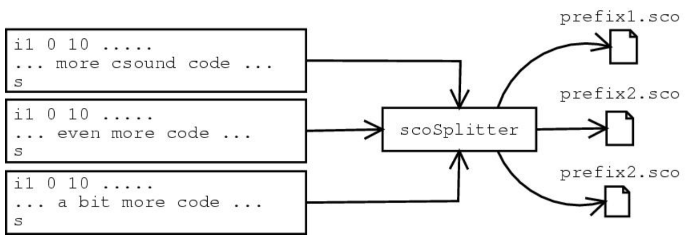
Figure 1: Overview of the process

and named manually. This is a hassle, especially if the composition changes and new sections are created between the existent ones.

2.2. Automatic splitting

In the previous sections we could see how handy and flexible is the use of Make to render Csound scores. The major problem is the somewhat cumbersome management and editing of separated files in large compositions. In this section will be presented a few solutions using the same concept but now the composer will be dealing with one score file only. The secondary files will be generated automatically.

Using the section command `s`. Probably the most direct aproach is the creation of a script that will read the score and create one file for each section defined with `s`. Thus, the problem of managing multiple files is solved, they are created automatically and named according a given prefix. The sections then can be compiled with the command `make prefix sectionNumber`. (fig. 2)



```
i1 0 10 .....
... more csound code ...
s
```

```
i1 0 10 .....
... even more code ...
s
```

```
i1 0 10 .....
... a bit more code ...
s
```

Figure 2: Splitting the score

Using a new section command. Although the previous solution is a great improvement, the mixer data is still separated from the music. A more complete solution is to create a new command `section`. It accepts a label and the start time of the section. Naturally Csound doesn't have this command and we are not going to implement it in Csound's core. The previous script will be modified to extract the sections using the new command. Since we want to have some backward-compatibility the `section` command will be preceded of the comment character `;` and the `|` character. The later is necessary to avoid the script catching a valide comment with the word "section". The ex. 2.4 shows the syntax. The lines 1 and 4 define valid sections while in line 7 we have a regular comment.

Now not only the files for each section are automatically generated but the mixing data as well. A good side effect of this approach is that the user no longer uses the mixer commands directly anymore. The mixing engine can be replaced without the user notice.

This is the best of the four solutions, backward compatibility is kept while gives more power, flexibility and convenience.

Example 2.4 The `section` command

```
;|section foo 0
i1 0 10 ...
....  more notes here ....
;|section bar 10
i1 0 10 ...
....  more notes here ....
; section, sweet section
i1 0 10 ....
......
```

3. From the concept of sections to events

3.1. Introduction

The previous solutions splitted the score in sections to recompile only the necessary parts. However, this procedure is inefficient when the goal is to manipulate smaller elements. In fig. 3 the boxes represent events in time. The common aproach of moving events around while composing is cumbersome with plain Csound because the durations of notes have to be recalculated manually, one by one. Splitting the score into sections, as in the previous examples, would not work since the large box in 3 delimits a section. The best solution is the possibility of defining *events*.

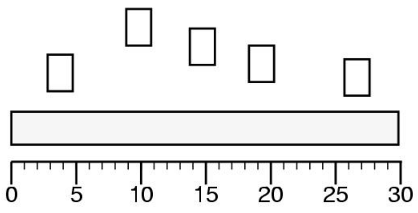


Figure 3: Events

I am working in a compositional environment called monochordum that uses Csound as a rendering engine. It implements, among other things, the concept of events. Monochordum works basically using the same idea of the previous examples; it creates a separated score for each event and a Makefile to control the rendering process.

The ex. 3.1 shows the basic syntax to create events. An event is defined with the `event` command followed by a label. The start time of each event can be defined with the option `-start`. The Csound code is inserted after `event` and the label, between curly braces. The option `-gain` determines the event's gain value in the final mixing. (fig 1)

Naturally, events can be nested. At that point it is possible to represent hierarquical and more complex structures than with plain Csound, or even with the previous section-based solutions.

3.2. Advanced features

In monochordum an event time can be expressed in relation to the time of other events. The implementation was freely based in the relations proposed by Allen (Allen 1991). This relations can indicate that one event starts after another event, or one event starts with another one. In ex. 3.2 the event “bar” starts right after “foo”, while the event “chords” starts at the same time of “bar”. The fig. 4 shows a graphic representation of ex. 3.2.

Example 3.1 Event syntax

```
event foo {
  i1 0 2 ...
  ...
}
event bar {
  i1 0 3 ....
  ...
}
foo configure -start 0 -gain .5
bar configure -start 30
```

Example 3.2 Events relations

```
foo configure -start 0
bar configure -start {after foo}
chords configure -start {with bar}
```

Events definitions can be more flexible if they have paddings (positive or negative). In ex. 3.3 both events "bar" and "chords" start after "foo", however "bar" has a padding of 2 seconds while "chords" a padding of -2 seconds. (fig. 5)

Others relations such as `before`, `finishes`, `middle` and `meets` are available, but the user can create their own relations as well.

4. Implementation

Monochordum is implemented in [incr tcl], a well-known object-oriented extension to Tcl. Besides the general organization of classes in attributes (e.g. `start` and `duration`) and methods (e.g. `with`, `before`, and `after`) some features of [incr tcl] are used such as the `configure` command. The value of an attribute can be changed with the command `object configure -attribute value`, where object is the name of the object, attribute is the name of the attribute, and value is the new value of the attribute.

5. Future work

This idea can be extended to work with other languages besides Csound. This will allow a more easy and consistent use of languages with different paradigms. A feature already implemented in monochordum is a high-level score language, capable of using notes names or letters (e.g. `do` or `c`), rhythms, chords, graphical notation (using *lilypond*), etc.

6. Conclusion

The score splitting into smaller files and the use of a recompilation tool such as Make can provide more flexibility, speed, and power in the process of Csound rendering. Neverthe-

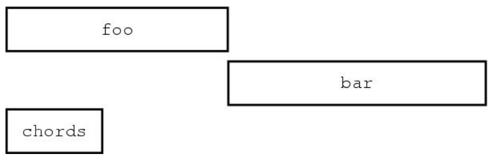


Figure 4: Events relations

Example 3.3 Events padding

```
bar configure -start {after foo} -pad {2s}
chords configure -start {after foo} -pad {-2s}
```

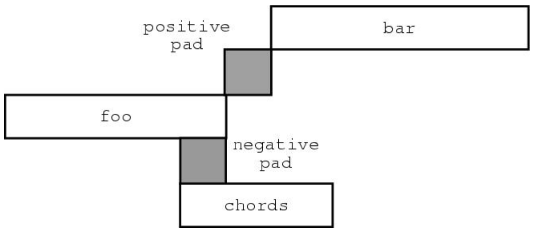


Figure 5: Events padding

less is necessary to create some sort of device to split the score and generate the secondary files automatically. The ultimate solution is the definition of events, a structure between a note list and a section. Besides solving elegantly the former problem, the definition of events make possible the creation of scores with a hierarquical structure. This concept has been implemented in monochordum, a compositional environment for Csound.

References

Allen, J. F. (1991). Time and time again: the many ways to represent time. *International Journal of Intelligent Systems 6*(4), 341–355.

Stallman, R. M. and R. McGrath (1998). *GNU Make: A Program for Directing Recompilation.* Boston, MA: Free Software Foundation.