An Experimental Classification of the Programming Patterns for Scheduling in Computer Music Programming

Hiroki NISHINO
NUS Graduate School for
Integrative Sciences & Engineering,
National University of Singapore
g0901876@nus.edu.sg

ABSTRACT

How to schedule a desired temporal pattern is one of the most elementary issues to consider when implementing a computer music system, and there already exist several major programming patterns for scheduling. However, such computer music-specific programming patterns seem to not be discussed as frequently as general programming patterns, and thus there may still be some necessity for additional clarification.

For instance, the programming pattern called *temporal recursion* may be better described as *self-rescheduling*, when contrasted with other programming patterns that perform similar tasks. In this paper, we describe four programming patterns that can be seen in the existing computer music languages and propose the names for these programming patterns. Such a discussion can benefit by initiating the discussion on the computer music-specific programming patterns in our community, to avoid an unnecessary ambiguity in further investigation of the related programming patterns.

1. INTRODUCTION

As computer music is essentially a time-based art, it is inevitable to consider how to realize desirable temporal behaviour when implementing a computer music program. Even when coding a simple program that only repeats a prepared phrase composed of a few notes, one must realize such temporal behaviour by scheduling each event at its own expected timing. Further labour would be required to perform more complex musical tasks, especially when multi-tasking must be involved.

Many computer music programming languages or software frameworks have been designed to support domain-specific needs for computer music applications; yet, while it can significantly reduce the effort made by a programmer in comparison with the effort required when writing a computer music program from scratch, its software design may also give certain constraints as to how such temporal behaviour of a musical task should be programmed, depending on the design of the language.

Such a discussion on the programming patterns[1] with respect to the temporal behaviour seems still unpopular in the computer music community. However, as live-coding [4] suggests, recent creative musical practices often involve some programming patterns with respect to time to a significant degree; unlike in the earlier decades when only expert computer music programmers dealt with such programming issues, even computer musicians without expert programming skills must face the same issues today. Considering such situations of our time, it is desirable to make some effort to classify the existing programming patterns to support further sound discussion. In this paper, we describe an experimental classification of several existing programming patterns in textual computer music languages.

2. TWO MODELS FOR SCHEDULING

We first classify how the scheduler is involved in a programming pattern into two different models: *explicit-scheduling* and *implicit-scheduling*. While many computer music languages and frameworks are indeed capable of both models of scheduling, it is beneficial to provide such technical terms for further discussion of the programming patterns, as it can directly influence the resulting implementation.

2.1 Explicit-scheduling

In some computer music languages and frameworks, a user program is expected to explicitly use the APIs provided in the programming environment for scheduling a task or an event at the desired timing.

For instance, *Impromptu* [13] and *SuperCollider* [16][2] are languages of this kind. In Impromptu, the '*schedule'* function is used to schedule a call to the function by giving it as an argument, together with the timestamp. SuperCollider provides several different objects for scheduling (e.g., '*SystemClock*', '*AppClock',* and '*TempoClock'*) which can be passed a '*Routine'* object or a *'Function'* object to be executed at the specified timing.

We propose *explicit-scheduling*, as the name for this method of scheduling, as the scheduler is visible even at

[1] In [10], Riehle and Züllighoven explain *programming pattern* as "a pattern whose form is described by means of programing language constructs", which is also "based on programming experience", and "we use these patterns to implement software design".

[2] Functions are first-class citizens in both Impromptu and SuperCollider.

the surface level of the code and a user program accesses its feature explicitly.

2.2 Implicit-scheduling

On the contrary, in some other computer music languages, the underlying schedulers may not be directly visible at the user program level. For instance, in LuaAV [14], its *wait* method yields the current coroutine and asks the scheduler to resume it again after the given duration or when a certain event occurs. In a strongly-timed programming language, such as ChucK [15] or LC [9], the assignment to the special variable '*now*' will suspend the current thread and the underlying scheduler resumes the thread at the given timing. In such languages, users are indeed implicitly utilizing the scheduler in the underlying software framework, while it seems just as a simple function call or an assignment at the surface level of the user code.

We propose *implicit-scheduling* for this manner of scheduling, in contrast to *explicit-scheduling*, as the underlying scheduler is not directly visible at the user program level.

3. PROGRAMMING PATTERNS FOR SCHEDULING

3.1.1 Temporal loop

Implicit-scheduling may be inserted within a looping control structure, interleaved between the sub-tasks in a computer music program. We propose '*temporal loop pattern*' for the name of this programming pattern. While it seems simple and trivial as a programming pattern, giving a name to the programming pattern is valuable even just to distinguish it from the other programming patterns related to computer music programing.

```
01: // synthesis patch
02: SinOsc foo => dac;
03:
04: // infinite time loop
05: while(true)
06: {
07:   // randomly choose a frequency
08:   Std.rand2f(30, 1000) => foo.freq;
09:   // advance time
10:   100::ms => now;
11: }
```

Figure 1. A simple strongly-timed program in ChucK [15, p.43].

Figure 1 [15, p.43] is a typical example of *temporal loop pattern*, often found in ChucK programs. As shown, *implicit-scheduling* is inserted within a loop structure to realize a desired temporal behaviour.

3.1.2 Repetitive-scheduler

In some computer music languages, the API for *explicit-scheduling* may have the features for repeatedly scheduling a task given as an argument. We propose the name, '*repetitive-scheduler*', for this programming pattern. Figure 1 describes a simple example of this programming pattern in SuperCollider, which is taken from its help file [2]. In the Figure 2 example, the *SystemClock* object and its *sched* method is utilized. As described, the *SystemClock.sched* method reschedules and executes the given function repeatedly, when the function returns a float value, interpreting it is duration before the next occurrence. Returning *nil* will stop this repetitive scheduling.

```
*sched(delta, item)
The float you return specifies the delta to
resched the function for. Returning nil stops
the task from being rescheduled

        SystemClock.sched(0.0, { arg time;
          time.postln;
          rrand(0.1, 0.9)
        });

        SystemClock.sched(2.0, {
           "2.0 seconds later" .postln;
           nil
        });
```

Figure 2. An example that utilizes *SystemClock.sched* method call in SuperCollider [2].

3.1.3 Temporal-recursion

It is also often possible to write a function so that it can reschedule itself again. Unlike the *repetitive-scheduler* pattern described above, in which the scheduler itself repeatedly schedules the same tasks, it is the callee function itself that is responsible for scheduling in this programming pattern.

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

(periodic (now) 0)
```

Figure 3. An example of *temporal-recursion* as Sorensen *et al*. describe in [13].

As it is discussed in [12], while a significantly similar programming pattern was already presented in the MOXIE [3] language and the CMU MIDI toolkit [5] in earlier decades, it is sort of 'rediscovered' by Sorensen, who developed the *Impromptu* computer music language. Sorensen and his colleagues named this programming pattern *'temporal recursion'*. Figure 3 describes an example of temporal recursion given in [13]. As shown, this programming pattern calls the API for rescheduling the function itself and thus involves *explicitly-scheduling*.

4. DISCUSSION

So far we described three programming patterns frequently seen in the existing computer music languages. While the *temporal loop* pattern involves *implicit-scheduling*, the other two involve *explicit-scheduling*.

One question we would like to raise at this point is whether *'temporal recursion'* is really appropriate for the programming pattern as Sorensen describes, when considering which is the better classification. While the definition of the function '*periodic'* in Figure 3 is recursive in that the definition of the function refers to the function itself, the function does not make a direct recursive call to itself – it asks the scheduler to reschedule itself.

We would like to propose another example for further discussion. The Figure 4 example is a recoded version of the Figure 1 ChucK example. As shown, the main loop is replaced with a recursive function call. Unlike the Figure 3 example in Impromptu, the example is based on *implicit-scheduling* and does not involve any instance of the scheduler at the surface level of the code.

Moreover, the Figure 4 example performs a direct recursive call within the function itself, as seen in many well-known recursive examples, such as the Tower of Hannoi and the Fibonacci number [6]; it is clearly a very simple example of recursion. *Tail-call optimization* [8, p.58] can be also safely applied to avoid wasting the frame stack.

```
01: // synthesis patch
02: SinOsc foo => dac;
03:
04: //recursion, instead of an infinite loop.
05: fun void recur(){
06:   // randomly choose a frequency
07:   Std.rand2f(30, 1000) => foo.freq;
08:   // advance time
09:   100::ms => now;
10:   recur(); //make a recursive call
11: }
12:
13: //call recur() to start the temporal recursion.
14: recur();
```

Figure 4. A simple strongly-timed program in ChucK, recoded with recursion.

When classifying these programming patterns, it would be desirable to contrast the related programming patterns as much as possible; in the earlier sections, we proposed the terms, *explicit-scheduling* and *implicit-scheduling* for this purpose, aiming to support further clarification regarding the difference in the scheduling models among these programing patterns.

Form this point of view, the Figure 2 example and the Figure 3 example are the programming patterns that belong to the *explicit-scheduling* group, and the Figure 1 example and the Figure 4 example belong to *implicit-scheduling*. One might note that the programming languages that involve *implicit-scheduling* indeed include a statement that causes the passage of the time within themselves, while the other programming patterns that utilize *explicit-scheduling* do not enclose any statement to invoke the passage of the time; when utilizing *explicit-scheduling*, the part of the tasks related to the passage of the time looks as if it is performed within the underlying scheduler, not within the user code. In other words, the Figure 1 and 4 examples clearly include '*temporal*' behaviour within the programming patterns, whereas Figure 2 and 3 examples exclude it.

In addition, the programming pattern in the Figure 3 example by Sorensen may be more similar to the *continuation-passing style* [1], as the programming pattern passes where the computation should continue to the scheduler, and it is the scheduler that invoke the given function.

Considering such an issue, it may be more appropriate to call the programming pattern in the Figure 3 example '*self-rescheduling'* rather than '*temporal recursion'*. Possibly, when considering the contrast between the recursion and the loop control structure, it may be better to call the programming pattern as seen in the Figure 4 *'temporal recursion'* instead.

It seems also beneficial to consider whether the function itself performs repetition or not. In this sense, the Figure 1 and Figure 2 examples belong to the same category, whereas the Figure 3 and Figure 4 examples belong to the opposite category; the latter group schedule the next occurrence of the functions explicitly within themselves, while the next occurrence of a task is controlled externally by a looping control structure in the former group.

		Where does the passage of the time seem to occur mainly?	
		implicit-scheduling	***explicit-scheduling***
		within the user code	within the scheduler
What controls the repetition?	The repetition is realized by the function to play the pattern itself ***internally***	***Temporal Recursion*** *(as we propose the name for the Figure 4 example)*	***Self-Rescheduling*** *(a possible new name for Sorensen's 'temporal recursion', shown in Figure 3)*
	The repetition of a pattern is performed ***externally*** by a loop control structure or the underlying scheduler	***Temporal Loop*** *(as described in the Figure 1 example)*	***Repetitive-Scheduler*** *(as described in the Figure 2 example)*

Table 1. An Experimental Classification of the programming patterns for scheduling in computer music programming.

By summarizing the above discussion, a 2D matrix to classify these four programming patterns can be drawn. One axis is *explicit-scheduling* or *implicit-scheduling*. The other axis categorizes whether the repetition is controlled internally or externally within the part of the code to play a certain pattern. In the examples in Figure 3 and Figure 4, the functions to play patterns internally reschedule themselves to the repetition. On the other hand, in Figure 1 and Figure 2, the repetition is realized externally by a loop control structure (Figure 1) or by the underlying scheduler of the software framework (Figure 2).

Thus, a matrix to classify these four programing patterns can be made as in Table 1. One may notice that the word '*temporal'* is used for the programming patterns that involve *implicit-scheduling,* while the word *'schedule'* is used to name the other patterns that involve *explicit-scheduling.* One view that possibly justifies such naming (and renaming of Sorensen's '*temporal recursion'* to

'*self-rescheduling'*) is that the behaviour of the programming patterns that utilize *implicit-scheduling* seems to involve the passage of time within, as each thread is actually suspended (or seems conceptually suspended at least at the surface level of the code), regardless of the actual implementation.

It should be noted that how a user stops scheduling can differ with the patterns. When using the self-rescheduling pattern, a user often redefines a callee function so that it does not reschedule itself further. Instead, a user often simply kills the thread when a temporal loop pattern is used. In addition, how the repetition of a phrase is terminated can also differ. In the former example of self-rescheduling, as the termination is achieved by redefinition of a callee function, all the sounds scheduled in the previous call are played by the scheduler, while in the latter example of a temporal loop, theu can be immediately terminated when the thread is killed; thus, while these patterns seem similar in functionality, the actual behaviour in practice can differ.

5. CONCLUSION

In this paper, we described four programming patterns in total, which are frequently utilized in computer music programming and then proposed an experimental classification of these programming patterns. The classification is based on two factors: (1) how the scheduling is performed (implicit-scheduling/explicit-scheduling) and (2) what controls the repetition (the user function itself/the external loop structure or the scheduler). Based on the discussion, we proposed names for these programming patterns. The discussion also led to a suggestion that one of the programing patterns (often seen in Impromptu) should be referred as '*self-rescheduling'*, instead of *'temporal recursion'* as Sorensen describes in [13], and that the name *'temporal recursion'* would be suitable for another programming pattern.

However, the aim of this paper is not to argue that these names are canonical. It is rather intended to invoke some attention to the necessity for the discussion of computer music-specific programming patterns, as seen on more general programming patterns among the programmer community; the name for the programming patterns should be given after collaborative and creative discussion in the community.

6. FUTURE WORK

While each programming pattern described in this paper performs a very simple and similar task, it can be expected that we would find more programming patterns if we closely observe actual computer music programming activity. For instance, live-coding performers may provide a number of interesting examples that perform more complex musical tasks. It would be beneficial also to discuss their programming patterns, especially because unlike in normal programming activities, live coding performers must write and modify their programs on-the-fly, on stage; they might take special care in the coding strategy so that they can perform desired tasks in a manner that is more suitable to such an abnormal programming situation.

7. REFERENCES

[1] A.W. Appel and T. Jim, "Continuation-passing, closure-passing style," In *Proc. the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL,* 1989.

[2] J. McCartney *et al.* "SystemClock" *SuperCollider 3.2 help files*. http://doc.sccode.org/Classes/SystemClock.html [Accessed on Apr/15/2014]

[3] D. J. Collinge, MOXIE: a language for computer music performance. In *Proc. of ICMC,* 1984

[4] N. Collins *et al.*, *"Live coding in laptop performance"*, *Organised Sound, Vol.8 (3), Cambridge University Press,* 2003.

[5] R. B. Dannenberg, *The CMU MIDI toolkit, version 3*, 1993

[6] L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, 1989

[7] J. M. Hoc *et al., Psychology of Programming.* Academic Press, 1990.

[8] R. Ierusalimschy. *Programming in Lua. The Third Edition*, Lua.org, 2013

[9] H. Nishino *et al.*, "LC: A New Computer Music Programming Language with Three Core Features", *submitted to Proc. ICMC-SMC,* 2014

[10] D. Riehle and Heinz Züllighoven. "Understanding and using patterns in software development." *TAPOS* 2.1, 1996, pp.3-13.

[11] C. K. Roy *et al.*, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sciences of Computer Programming, Vol. 74 (7)*, 2009. Pp.470-495

[12] A. Sorensen, *The Many Faces of Temporal Recursion, http://extempore.moso.com.au/temporal_recursion.html*, 2013 [Accessed on Apr/15/2014]

[13] A. Sorensen *et al., "Programming with time: Cyber-physical programming with Impromptu"*, in *Proc. ACM SPLASH/OOPLSA,* 2010

[14] G. Wakefield *et al., "LuaAV: Extensibility and heterogeneity for audiovisual computing", in Proc.* the Linux Audio Conference, 2010.

[15] G. Wang, *The chuck audio programming language. A strongly-timed and on-the-fly environ/mentality*. Ph.D thesis, Princeton University, 2008.

[16] S. Wilson *et al, The SuperCollider Book.* The MIT Press, 2011.