# Generative Music Using Reactive Programming

**Bertrand Petit**
Inria, Sophia Antipolis, France
bertrand.petit@inria.fr

**Manuel Serrano**
Inria, Sophia Antipolis, France
manuel.serrano@inria.fr

## ABSTRACT

*Generative music,* i.e., *music produced using algorithms or assisted by algorithms, can be created using may different techniques and even methodologies. It can be generated from grammatical representations, using probabilistic algorithms, neural networks, rule-based systems, constraint programming, etc. In our work, we are interested in a new technique that combines complex combination of basic musical elements with stochastic phenomena, and that is made possible by the use of synchronous reactive programming. We have based our work on the HipHop.js programming language that allows composers to create music programs and that produces satisfying and unexpected musical results. In this paper, we present this new way of composing music and we comment some concrete realizations.*

## 1. INTRODUCTION

Generative music or algorithmic music is a discipline that dates to the 50s and the origins of computers. Today, it has turned into an broad field of research that uses a wide range of computer technologies such as grammatical representations, probabilistic methods, neural networks, rule-based systems, constraint programming, etc. In this article we describe a system based on the HipHop.js [1] synchronous reactive programming language, a model invented in the 80s for programming complex automata has found in critical systems[2]. We use this system to evaluate the relevance of a generative music production technique based on automata, synchronous programming, and random phenomena.

In Section 2, we briefly presents generative music. The basic concepts of our generative music system are presented in Section 3. We briefly comment its implementation in Section 4. Section 5 presents synchronous programming with HipHop.js that is illustrated by a simple example in Section 6. We discuss two major musical aspects of our system in Section 7. We present the future work in Section 8 and we conclude in Section 9.

## 2. GENERATIVE MUSIC

Composition based on algorithms has been an active domain since the early days of computing. A good survey can be found in an article by Fernandez and Vico [3]. It deals with the state of the art of Artificial Intelligence techniques applied to musical composition and, in general, to algorithmic composition. For French-speaking readers we recommend the work of Christophe Robert [4] who presents the subject in a synthetic and educational way. According to Ch. Robert classification, our work falls in the category of self-organized combinatorial systems and stochastic systems. Stochastic systems were widely studied by the composer I. Xenakis in the 1960s [5]. They are still used in the form of Markov chains by the composer Philippe Manoury for example. The combinatorial systems, in line with our work, have a long history dating back to the Wüfelspiel of the 18th century [6] and to Kircher's Organum Mathematicum [7]. The works of François-Bernard Mâche such as Maponos [8] are among more recent reflections in the field of combinatorial music. The "duo Auterecre" produces other examples of combinatorial music, but based on permutations. We will see that our work follows the same approach but adds a decisive technological element: the programming of reactive systems.



## 3. BASIC CONCEPTS

Our system, called Skini, generates music by *sequencing patterns* (or *musical characters*). These patterns are organized in *sets* according to *scenarios*, that react to *stochastic phenomena*. These components are described in this section.

### 3.1 Pattern

Patterns, or musical characters, are sound elements produced by a composer. They can be MIDI sequences, sound samples, or even short scores. Our objective is to produce musical works that will be combinations of these patterns and that can, in addition to their combination, undergo treatments. In the case of MIDI sequences, it will be possible, for example, to perform transpositions, retrogradations or inversions. In the case of sound samples, filtering or another signal processing can be carried out. The durations of these patterns are important parameters of the system on the speed of its evolution, and on the complexity of the combinations. We generally use short patterns that last a few seconds because it gives flexibility to the system, but longer patterns could be used too.

### 3.2 Set of patterns

Set of patterns reflect the actual constraints of actual instruments, as a real instrument can only play one pattern at a time. Using set of patterns greatly reduces the number of possible combinations. For instance, a set of $N$ patterns yields to $2^N$ possible pattern combinations. That is, for a

base of 10 patterns we have 1024 combinations, for 50 patterns $10^{15}$, etc. For $N$ patterns divided into three groups of I, J and K elements each with $N = I + J + K$, using set of patterns reduced the number of possible combinations from $2^N$ to $2^I + 2^J + 2^K$. For instance, for 10 patterns and three groups of 2, 3 and 5 elements it reduces the initial 1024 combinations to only 44.

### 3.3 Stochastic Engine

The stochastic engine of Skini consists in randomly choosing patterns among the sets of patterns which are made visible (we will also say *activated*) according to a *scenario*. When the engine chooses a pattern, it generates an *event* in Skini. We called this choice and event generation, *selection*.

### 3.4 Scenario

A scenario makes the patterns organized in sets visible to the stochastic engine. It is implemented as a computer program that reacts to events that can be clock events, selections of patterns, selections of pattern among specific sets, random values, etc. The scenario controls the combinations of sets and organizes the sequences of these combinations to produce coherent and consistent music.

The patterns are played by *instruments*, synthesizers, or musicians. We assume that an instrument can only play one pattern at a time. This sequencing mechanism is necessary to ensure a good independence between the operation of the stochastic engine and the scenario.

## 4. IMPLEMENTATION

The architecture of the system is presented in Figure 1. It consists of three layers. At the top, the *music production* layer generates the final musical result. The music is generated by a Digital Audio Workstation (DAW) that is fed with the patterns stored in a database. The middle layer runs the *scenario* and the *sequencing* of patterns according to the stochastic engine which select the patterns to be played. The bottom layer is for the stochastic engine. Note that this architecture can be adapted to interactive music production by replacing the bottom layer with an audience interaction layer.

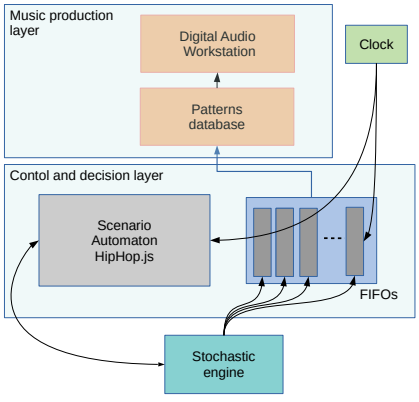


**Figure 1**. Architecture

### 4.1 Scenario Design

If the design of patterns and the organization of pattern sets requires musical skills essentially, scenario design requires technical skills. In our system, the scenarios triggers actions according to successive or random events. This type of control is managed by *automatons*. For music to be generative, *i.e.,* the result of an algorithmic process, these automatons must manage complex situations. The programming of complex automata is known to be a delicate topic that is the subject of many formalisms such as Petri's networks [9], Grafcet[10], UML activity diagrams, or synchronous languages [11]. But before tackling programming, the composer can start by graphically representing the possible paths of the pieces generated by the system, much as he would draw the large structures of a score.

Figure 2 is an example of a representation of 3 sessions in a scenario. Each session may be triggered according to a random process, depending on time execution or the occurrence of one or more patterns. This allows a single scenario to yield to radically different generated music.

In this representation pattern sets are represented by their names (ViolinScale, ViolasScale...). Vertical lines correspond to the activation and deactivation of the sets. Arrows between these lines are associated with sets. When a number $N$ is associated with an arrow it means that the automaton waits until $N$ occurrences of patterns belonging to the set have been requested by the engine before moving to the next step. For instance, in the "Tonal session", the automaton will activate the second sets after 5 occurrences of "ViolinsTonal". A single line means that any patterns of a set can be selected by the stochastic engine. This graph also used a particular type of set that are called *tank*. They are noted in blue rectangles. These are sets containing patterns that can be activated only once by the stochastic engine. Using tanks limits the number of repetitions of the same pattern and avoid monotony of the generated music. The dotted lines refer to the measurement of time. For example, in the "Scale session" the "TrumpetsScaleTank", "HornScaleTank" and "TromboneScaleTank" tanks are deactivated after 20 ticks.

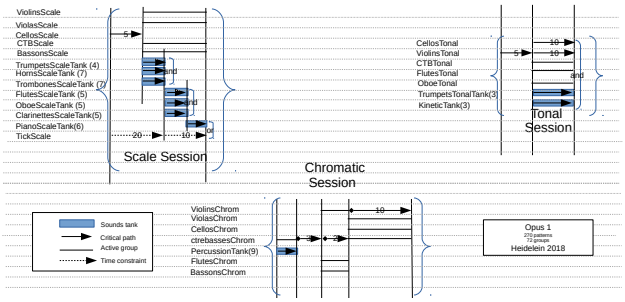


**Figure 2**. An example of scenario

This graph also includes constraints in between pattern sets. The Scale session will only activate the piano when the three "FlutesScaleTank", "OboeScaleTank" and "ClarinetsScaleTank" are empty. This representation is not as precise, as an actual program executing the final scenario could be, but it sketches the shape of the music to be generated.

The degrees of freedom, left to a random process of selection, are expressed macroscopically with this representation. In addition to this representation of activation and deactivation of sets, the composer will be able to add random phenomena on tempi, on pattern processing processes such as transpositions for example. It may also define rules for combining different 'sub-scenarios'. The particularity of this way of producing a scenario is that the composer can keep control over the combination of patterns and the overall coherence of the music. The scenario definition leads quickly to a complex automaton. And we have found that it's very difficult to think of all the combinations that a scenario can create. We will see a little further how to efficiently create scenarios by means of an adapted programming language, from the family of synchronous reactive languages.

### 4.2 The stochastic engine operation

The stochastic engine selects randomly and regularly patterns amongst those activated by the scenario automata and can, to some extent, modify the sequence of selections.

Globally, the patterns selected by the engine are transmitted to the DAW (or any other sound system) for being played or recorded. The scenario automata is informed about the patterns that have been selected by the engine. This information impacts the next steps the scenario automata will executed.

More precisely, the behaviour of the stochastic engine is controlled by two main parameters in order to avoid system congestion and to control the speed at which the music may evolve: a variable *delay*, that defines the rate at which patterns are requested, and a *waiting time limit* . The *delay* corresponds to the rate at which the patterns are selected and entered in the FIFO (see figure 1). It is defined by a minimum and a maximum value. According to these values, the stochastic engine randomly chooses the time at which it requests the next pattern to be played.

The *waiting times* limit is a consequence of the fact that *an instrument can only play one pattern at a time*. If, for the same instrument, the engine requests a B pattern before a previous A pattern is completed, we risk losing part of the A pattern. This is why we have set up a queuing mechanism. Each instrument has its own queue (FIFO). A selection by the engine adds a pattern execution request to the queue that is destacked at a regular rate. This stacking mechanism generates durations before a pattern is played. The stochastic engine takes into account these durations because they may have a strong impact on the behaviour of the musical piece. For example, if the scenario allows the selection of in a set of patterns for a fixed period, allowing to select pattens without limit could lead to 'saturate' the FIFO of the instruments corresponding to the set.

The stochastic selection process is simply random. Nevertheless, the engine includes an algorithm that avoids successive repetitions of the same pattern. Its principle consists in memorizing the last three selections for each instrument, and to preferably select a pattern which has not been selected during the last previous three selections. This has a clear impact on the generated music when the number of patterns in a set is low. In this case random selection algorithms can easily repeat the same pattern.

## 5. SYNCHRONOUS PROGRAMMING

In the 80s and 90s, three languages adapted to automaton programming appeared in France: Esterel [2], Lustre, and Signal [11]. Lustre and Signal have been designed for the processing of data flows. Esterel was designed for flow control. Esterel and Lustre were marketed by "Esterel Technologies" in an integrated environment called SCADE (Safety Critical Application Development Environment). SCADE is used for critical systems in the field of transport or energy. Other languages have appeared following the initial trio, such as Lucid Synchronous [12] which process data streams like Lustre, or ReactiveC [13] and ReactiveML [14] in the Esterel lineage.

The HipHop.js language [1] that we use, is a dialect of Esterel meant for JavaScript. Its goal is to offer an efficient way to combine synchronous and asynchronous tasks inside a same program. A program written in HipHop.js is compiled in JavaScript and runs on unmodified JavaScript engines. HipHop.js programs are based on two basic principles.

- A HipHop.js program does not take *conceptual* time. That is, the language semantics ensures that the program behaves as if executed on an infinitely fast computer. As executions take (conceptually) no time, executions are synchronous. Of course, when executed on an actual computer, HipHop.js programs take an observable wall clock time but the languages semantics prevents programs to notice that actual physical duration, whatever constructs, *e.g.,* sequences, parallels, conditionals, or preemptions, they use.

- HipHop.js relies on signals that replace variables of mainstream programming languages. A signal has a status (emitted or not) and a value. A program can emit a signal, receive it, wait for it, or test its status. Most HipHop.js instructions deal with signals. HipHop.js programs have *input* signals that are associated with external events such as network events, timeouts, user interfactions, etc. The call of a HipHop.js program and the response it provides is called a *reaction*, hence the term reactive programming frequently used for Esterel/HipHop.js programming.

- Although parallel, HipHop.js programs are deterministic.

These principles enable the definition of instructions with clear semantics and therefore allowing the development of efficient and reliable compilers. The determinism of the executions helps developing and maintaining programs.

## 6. APPLICATION OF SYNCHRONOUS PROGRAMMING TO OUR PRINCIPLES

Let us consider a simple scenario session that could take place in an actual score. This session could first activate cellos, and violins playing staccato patterns. After five ticks (where ticks are defined by the music tempo), percussion patterns could be activated and then, after the first percussion patterns have been played, the cellos could be

played five patterns and stopped. In parallel violins should stop after 4 cello events or after 10 ticks.

This seems a mundane session and complete scenarios are expected to be populated with many similar components. However, although simple, the associated automata is already complex. Figure 3 that shows the time events of that orchestration, gives some hints on this complexity. Traditional programming languages are not well suited for implementing automaton because as the automaton states have to be implemented with variables and functions, a mere change in the specification generally requires a whole re-programming. HipHop.js has been designed for solving that problem. It implements automaton with higher level constructs that can be composed one with another. Changing the specification of the automata then only requires local changes to the program.

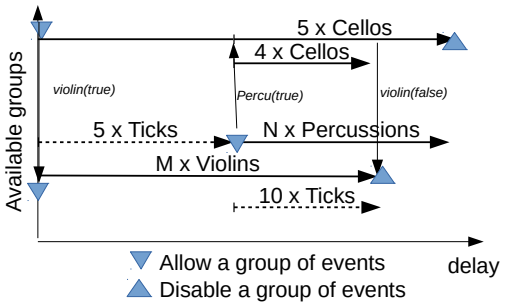


**Figure 3**. Simple example.

A HipHop.js execution is split in a succession of *reactions* that are triggered by external events generated by the stochastic engine. A HipHop.js program is organized as a list of modules that are loaded into a *reactive machine*. A module specifies the signals it can receive and emit. The module implementing our example is as follows:

```
hiphop module orchestration(
    in Tick, out Perc, in PercPlaying, out Cello,
    in CelloPlaying, out ViolinStacatto) {
   emit ViolinStacatto(true);
   emit Cello(true);
   await count(5, Tick.now);
   emit Perc(true);
   await (PercPlaying.now);
   fork {
      await count(5, CelloPlaying.now);
      emit Cello(false);
   } par {
      await count(4, CelloPlaying.now)
            || count(10, Tick.now);
      emit ViolinStacatto(false);
   }
}
```

The module `orchestration` first emits two output signals `ViolinStacatto` and `Cello` with a value `true`. Then, it waits for five ticks before emitting the signal `Perc` with a value `true`. These emissions will be received by the automaton that will modify the status of the scenario accordingly. The value `true` means that the patterns in a set of patterns (here Cello, Perc and ViolinStacatto) is activated and can be selected by the stochastic engine. A value `false` does the opposite. It deactivates a set of pattern.

The `fork/par` control structure runs its branches in parallel and waits for all of them to complete. The branches can communicate and synchronize with each other by broadcasting signals that are delivered instantly. During a reaction, all branches of the all parallel constructions receive the same information. The HipHop.js execution is synchronous and deterministic.

HipHop.js can block on complex predicates. For instance, in the second branch of the `fork/par` construct, it waits for the first condition of 4 `CelloPlaying` events or 10 `Tick`. HipHop.js imposes no constraint on conditional expressions. They can mix arbitrary temporal expressions and JavaScript expressions.

This example, although simple, illustrates the benefit of HipHop.js. The constraints of the musical orchestration are mapped directly into constructs of the language without extra bookkeeping or encoding techniques. This greatly alleviate the composer/programmer task as it dramatically reduces the distance between the representation of his score and its actual executable implementation.

## 7. DISCUSSION

Skini presents two main concerns for a composer: the balance between determinism and stochastics, and the evaluation of the quality of the music.

Regarding the balance between determinism and stochastics, HipHop.js allows the composer to create scenarios whose results can range from relatively deterministic to completely random. According to the artistic project and the use that will be made of the work, the composer will have to define the extent to which his music should leave something to chance. If the scenario is very precise in its behaviour, the different instances of the work produced by different activations of the system will be close. A purely sequential automaton in time, for example, will only allow the choice of patterns in successive sets as a degree of freedom. If the number of patterns is not very high and if the patterns are similar, different executions of the scenario will give very similar musical results. Conversely, the design of a complex automaton can generate combinatorics, that are impossible for a human brain to design and can produce results that the composer/designer had not thought of at all.

Regarding the second concern and the evaluation of generative music, Skini faces the same problem as all musical generative systems. How to evaluate a musical result, which is essentially subjective? This is even more true for Skini, which is not a solution to reproduce a musical style, but a solution which proposes a specific pattern based composition method. Some Skini productions have been submitted to different people, experts or not in music. We have noticed that for music structurally based on patterns, such as most jazz styles for example, the results are convincing even with simple scenarios. We experiment that very linear constructions, such as pieces of tonal music, are less suitable for random constructions based on scenarios than other types of music (modal, serial, atonal...). For large-scale orchestral productions (symphony orchestra with hundreds of patterns) and complex scenarios, the system gives results where the music produced does not seem to be produced with the help of a computer.

As a demonstration, different music produced by the composer Heidelein using HipHop.js are available on SoundCloud [1] . The recording "Opus2-2-5-Instances" [2] is an example of 5 successive executions of the same automaton in loop. This example of an orchestral piece shows how several performances can produce very different results. This example is based on 191 patterns divided into 30 groups. The automaton code represents 800 lines of HipHop.js code.

## 8. FUTURE WORK

Programming using HipHop.js is powerful but requires computer science skills that music composers are not expected to have. Graphical programming could be a solution to this problem. As seen in Figure 2 scenario can be naturally described graphically, so it might be interesting and maybe not too difficult to design a graphical language to translate these graphical scenarios into synchronous programming. Graphical programming will impose limits on the complexity of automata, but it should allow non-computer-litterate musicians to use this type of technique.

When using stochastic techniques one must evoke I.Xenakis. The combinatorial and stochastic techniques he used, deal mainly with the production of off-time structures [5], i.e. basic materials for the design of a work than with the structure of the work itself. To make is short, while being globally in an approach that could be placed in the lineage of I. Xenakis, we have reversed some roles. In Skini, the off-time structure is musically conceived, and the in-time structure, i.e. the transformation into a score, is the result of a stochastic process. In Xenakis' work, the off-time structure is built from probabilistic models, and the score, therefore the final musical in-time gesture, is conceived by the composer. Compare to the stochastic processes of Xenakis, the stochastic engine we currently use is overly simple. It selects patterns randomly among those activated at a given time using random and bounded delays. More elaborate engines could be used. Without going so far as Xenakis in his book Formal Music [15], it would be interesting to understand the impact of different random processes, such as Markov chains, on the musical result by using them as the engine of our solution.

## 9. CONCLUSION

This paper presented Skini, a solution for generative music based on patterns and stochastic events. It follows the tradition of combinatorial techniques and patterns to produce music, which has a long history. Skini introduces an abstract dimension resulting from reactive programming. We have seen that this programming technique allows the creation of complex but coherent combinatorial and random scenarios difficult to imagine for a composer.

Since Skini can produce a musical structure in real time, one of its use cases is music for video games. Indeed, Skini could be used to feed scenarios with information from a real-time game. In the same way, by taking into account different events coming from an environment, Skini can be used to generate continuous and non-repetitive music in public location, such as museums, exhibitions or shops.

[1] https://soundcloud.com/user-651713160

[2] https://soundcloud.com/user-651713160/opus2-2-5-instances

**Acknowledgments**

Thanks to the composer François Paris and the CIRM who accompanied us throughout our research.

## 10. REFERENCES

[1] B. G. and M. Serrano, "HipHop.js: (A)Synchronous Web Reactive Programming," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, London, UK, 2020.

[2] G. Berry, "Esterel v7 for the Hardware Designer draft 1.1," 2008.

[3] J. D. Fernandez and F. Vico, "Ai methods in algorithmic composition: A comprehensive survey," *Journal of Artificial Intelligence Research*, vol. 48, pp. 513–582, 2013.

[4] C. Robert, "MusicAlgo," 2017. [Online]. Available: http://musiquealgorithmique.fr/

[5] Xenakis, "Vers une Métamusique," *La Nef*, vol. 29, pp. 117–140, 1967.

[6] F. Daxecker, "Der Jesuit Athanasius Kircher und sein Organum mathematicum." *Gesnerus*, 2000.

[7] P. Findlen, *Athanasius Kircher: The last man who knew everything*, 2004.

[8] F.-B. Mâche, "Trois chants sacrés," 1990. [Online]. Available: http://brahms.ircam.fr/works/work/10369/

[9] A. Choquet-Geniet and P. Richard, "Petri Nets," in *Software Specification Methods*, 2010.

[10] R. David, "Grafcet: A Powerful Tool for Specification of Logic Controllers," *IEEE Transactions on Control Systems Technology*, 1995.

[11] P. A. Laplante and S. J. Ovaska, "Programming Languages for Real-Time Systems," in *Real-Time Systems Design and Analysis*, 2011.

[12] J. L. Colaço, G. Hamon, A. Girault, and M. Pouzet, "Towards a higher-order synchronous data-flow language," in *EMSOFT 2004 - Fourth ACM International Conference on Embedded Software*, 2004.

[13] F. Boussinot, "Reactive C: An extension of C to program reactive systems," *Software: Practice and Experience*, 1991.

[14] G. Baudart, L. Mandel, and M. Pouzet, "Programming mixed music in ReactiveML," in *the first ACM SIGPLAN workshop*, 2013, p. 11.

[15] I. Xenakis, *Musiques formelles*, Editions R, Ed. Paris: Editions R, 1963, vol. 253 and 25.