AUTOMATICALLY GENERATING SYNTACTICALLY CORRECT AUDIO EFFECTS IN Max

Aron Glennon, Robert Rowe, Juan Pablo Bello

New York University
Music and Audio Research Lab (MARL)
{apg250, robert.rowe, jpbello}@nyu.edu

ABSTRACT

Creating novel audio effects in Max is a challenging task, involving both advanced knowledge of signal processing and the Max software environment. Automatic programming, which can teach a computer how to discover and produce novel effects on its own, may be used to accomplish this task. In order to best leverage automatic programming to this end, it is beneficial to have a system in place that can guarantee production of Max patches inherent with two key characteristics: syntactic correctness [4] and the ability to function as an audio effect. We present a system that automatically creates such Max patches using C++. We first introduce a set of classes that provide C++ with a sufficient internal representation of a Max patch. We then outline the process by which our system generates patch representations and discuss several rules that it follows in order to ensure syntactic correctness when translating these representations back into Max.

1. INTRODUCTION

The introduction of MSP to the Max visual programming environment in 1997 fundamentally changed the way many composers used Max. MSP introduced sampling, synthesis, and audio processing into the Max environment, providing the user with complete control over the creation, manipulation, and playback of audio [5]. The ability to create audio effects and synthesizers inside a Max window made the Max programming environment more powerful and consequently more popular.

While many publicly available effects processors made in Max (using MSP) mimic popular effects seen in rack mount hardware units, guitar pedals, and DAW software packages, the MSP library allows the flexibility to create useful novel effects as well. There are two reasons why this is a difficult task in practice: (1) developing compelling novel effects typically requires advanced knowledge of signal processing and (2) almost every imaginable straightforward combination of basic MSP objects has been used to develop effects over years of research in the field, leaving only effects made of complex object combinations left for discovery. Thus, the task of creating valuable novel effects using basic MSP objects is often beyond the reach of even the most experienced Max users. However, the issues outlined above can be circumvented if the task is approached through the lens of automatic programming [1].

The challenge of creating novel effects in Max can be thought of as searching through the space of all possible combinations of Max objects and connections for the subset that produce novel audio effects and then automatically generating them. This space is enormous even if patch depths are restricted and object arguments are limited to a discrete set of values. For example, the number of unique combinations of MSP objects and connections in a patch containing only thirty objects (with only one connection coming into and going out from each) is approximately $10^{35}$ and this doesn’t even take into account the allowance of different argument values for each object. The search space is tremendously reduced if restricted to syntactically correct Max patches that function as audio effects [4]. We have created a system that is able to automatically generate a patch of this type, which not only allows us to search through a much smaller space for novel audio effects, but also presents a way to correctly construct novel effects in Max when they are found.

2. REPRESENTING Max PATCHES IN C++

C++ must possess an internal representation of a Max patch’s contents and have an understanding of Max’s syntax in order for it to generate syntactically correct Max patches. To address the first requirement, we wrote classes in C++ to appropriately represent the structure of Max patches, objects, and connections. We considered a Max patch’s readability when designing these classes, introducing the concepts of object ‘level’ (vertical position in a Max window) and ‘level index’ (horizontal position) in order to specify the graphical placement of each object in a patch.
2.1. C++ Containers for Max Patch Components

We created the `MaxPatch` class to maintain a representation of a Max patch’s contents. Each object level in a patch is represented by a single vector of `MaxObjects`, whose level indices are determined by their position in the vector. A multi-level Max patch therefore requires a vector of vectors of `MaxObjects` (see Figure 1) to maintain representations of every object in the patch.

The `MaxObject` class contains a string representing an object’s given name in Max, a vector of arguments (each of type double), a vector of outlets (of type string), and a vector of vectors of `MaxInlets`. An `MaxInlet` is a struct containing a string that specifies an inlet type (the type the inlet expects) and sensible maximum and minimum values accepted at that inlet. Each inlet in Max can accept more than one type and therefore is represented in C++ with a vector of `MaxInlets`. Consequently, a full representation of an object’s inlets requires a vector of strings (of `MaxInlets`). However, a full representation of an object’s outlets only requires a vector of strings since a single Max outlet only outputs one type and neither maximum nor minimum value designations are applicable (see Figure 2).

The `MaxPatch` class also contains a vector of `MaxConnections` to represent the connections between objects in a patch. A `MaxConnection` is a struct that contains location information (level and level index) about the objects involved in the connection, the outlet of the output object, the inlet of the input object, and the type of information that flows over the connection (e.g. signal) (see Figure 3).

![Figure 1](image1.png)

**Figure 1.** The objects in a Max patch are represented in C++ using a vector containing vectors of `MaxObjects`.

**Figure 2.** A Max object’s inlet is represented in C++ as a vector of `MaxInlets`. An outlet is represented by a string specifying output type.

3. RESTRICTING THE SEARCH SPACE

3.1. Ensuring Syntactic Correctness

The classes and structs discussed in the previous section provide adequate containers for representing any Max patch. However, it is not guaranteed that any combination of these containers (i.e. `MaxObjects` and `MaxConnections` in a `MaxPatch`) will produce a syntactically correct patch in Max. Prior knowledge of the various objects available in the Max environment and the structure of each is also necessary. Therefore, as an initialization step, our system reads from a text file containing object-specific information about the objects that it will use to build patches. This information...
includes: an object’s name in Max; data on each of its inlets (types accepted, reasonable minimum, and maximum values); and data on each of its outlets (type returned). While specifying a reasonable range of values accepted at each inlet is not necessary for syntactic correctness, it ensures that objects are used as intended when each inlet receives values in its desired range and, therefore, further restricts the search space.

3.2. Using Root Trees to Represent MSP Audio Effects

In order to generate a patch intended for use as an audio effect, the patch must be representable as a black box with audio entering in one end and exiting out the opposite end. Thus, the patch must contain a signal that originates from an audio input object (e.g. \texttt{adc~}) and terminates at a \texttt{dac~} object, which sends the audio signal from Max to the computer’s audio output device.

The audio input object may not be the only object in the patch that generates data. If additional data streams exist, they will function as parameter drivers for the objects through which the main signal path flows. In contrast, multiple output objects cannot exist. All data streams must terminate at the \texttt{dac~}.

These properties suggest that a syntactically correct Max patch functioning as an audio effects processor can be represented by a directed root tree [3] with implied direction towards the root (i.e. \texttt{dac~}) (see Figure 4).

![Figure 4](image1.png)

Figure 4. An example Max patch represented as a rooted tree with direction flowing towards the root. The terminal objects are highlighted.

Our system uses this representation to build Max patches in C++. All patches are designed from the ‘bottom up’ starting with the \texttt{dac~} as the root of a tree and building up from the root one level at a time. The number of objects generated per level is equal to the total number of inlets at the previous level. Therefore, every inlet in the patch is involved in a connection. \texttt{loadmess} objects are used to supply other objects with the equivalent of arguments (rather than time varying parameters at every inlet). Replacing arguments with \texttt{loadmess} objects has the added benefit of being able to represent both arguments and objects with nodes in the root tree (thus making the root tree representation even more appropriate).

A Max patch will function as an audio effects processor on a given input signal if a path exists from the audio input object (found at one of the terminating points of a branch) to the \texttt{dac~}. Therefore, the only requirement of the patch during design is that at least one signal path continues from the \texttt{dac~} up through the current level being assembled, until an \texttt{adc~} is created to complete this path.

A number of different objects may be used to extend branches as long as the above requirement is met throughout design. These include the set of objects that do not take input (and therefore generate a data stream of some sort). The objects in this set are referred to as ‘terminals’ because they terminate the branch to which they belong [4]. The terminal set includes both \texttt{adc~} and \texttt{loadmess} objects.

As previously mentioned, the above strategy for creating Max patches results in directed rooted trees whose branches point to the root (known as reverse arborescences in graph theory [3]). This means that for each generated patch there will exist a unique path from the root to any other node (object) in the tree. This may seem limiting at first, because Max patches can (and often do) involve branches that cross, implying that a directed acyclic graph (DAG) may be more appropriate. However, any DAG with crossed branches can be rewritten without crossings as long as node duplication is possible (which is the case in Max) (see Figure 5):

![Figure 5](image2.png)

Figure 5. Two functionally equivalent versions of the same effect with (a) and without (b) crossed branches.

4. FROM ROOT TREES TO Max PATCHES

We represent a syntactically correct Max patch in C++ by filling a MaxPatch object with random MaxObjects and MaxConnections (generated using the rules and restrictions laid out in the previous section). Once a MaxPatch object has been sufficiently filled, its contents
must be translated to a language that Max understands: JavaScript.

4.1. Translation to JavaScript

The Max/js object was developed to allow users to create and connect objects and to interface with Max’s scripting architecture using JavaScript. A js object connects to a specific JavaScript file whose filename is input as the object’s first argument. The js object can function in an ‘autowatch’ state that enables automatic file watching so that anytime the currently loaded JavaScript file has been modified by another editor, js will reload and recompile the file.

Our system generates a file of JavaScript that outlines what objects to create, where to place them (based on object level and level index values), and what connections to make between them directly after C++ has produced a random, syntactically correct MaxPatch. A js object with autowatch running is connected to the JavaScript file so that its corresponding patch will be created in Max immediately after it is designed in C++.

The JavaScript generated by our system contains a delete method, which deletes the contents of the patch it is responsible for creating. This becomes useful if our system generates a large number of patches and we want to listen to how our input sounds through each patch. Instead of having to manually delete every patch before allowing the computer to create another, the delete method can be used in Max to delete the current patch and alert C++ that Max is ready for the next. Thus, the creation and deletion of concurrent patches becomes automated.

5. RESULTS AND DISCUSSION

The Max patches generated randomly using the methods discussed in this paper are syntactically correct and function as effects processors on an input signal. These patches form a small subset of the entire space of possible combinations of Max objects and the connections between them, while also including the even smaller subset of novel audio effects.

While any real-time Max audio effect can be represented using a reverse arborescence, there is an inherent bias towards generating patches with uncrossed branches. Though it is true that the functionality of a Max patch with crossed branches can be obtained in a Max patch with uncrossed, duplicate branches, the probability of exact duplication of a set of objects in two different parts of the patch is very low. Therefore, the probability of randomly generating any Max patch that is functionally similar to one with crossed branches is also low. To eliminate this bias, a non-negligible probability of the duplication of objects or sets of objects must be built into the system.

We introduce a further bias into the system by specifying reasonable minimum and maximum values accepted at each inlet. While this does further restrict the search space, it does so at the risk of excluding possibly useful objects, since interesting results can be found when using objects in ways in which they were not intended to be. Therefore, it is important to impose lax restrictions on an inlet’s bounds, so that they only safeguard against making an object’s output completely useless.

Even with these restrictions, most of the generated random Max patches are not interesting and often result in either silent or continuously clipping or singular valued-audio. Creating novel effects through automatic programming in Max is a complex task and the expectations of the work thus far were not to have solved it, but to have started moving towards a solution. The ability to automatically generate any syntactically correct Max patch that functions as an audio effect is a necessary foundation as it severely limits the search space.

Finally, it should be noted that this framework could also be used to generate new synthesis algorithms by replacing audio input objects with simple waveform or noise generators. Furthermore, the authors believe this framework could be used with slight modifications for a number of interesting automatic programming tasks in Max, not only limited to audio applications, but possibly also to video and physical computing.

6. FUTURE WORK

The next step towards automatically creating novel effects in Max is to find an optimized way to search through the space of all possible syntactically correct Max patches (that function as effects processors) to find those that meet certain novelty criteria. Defining criteria for novelty is a complex issue in the first place and one we will address before determining which search algorithm is most appropriate for this task.

7. REFERENCES