Cellular automata and music

Using the Java language for algorithmic music composition

Paul D. Reiners

May 18, 2004

Take computers, mathematics, and the Java Sound API, add in some Java code, and you've got a recipe for creating some uniquely fascinating music. IBM Staff Software Engineer Paul Reiners demonstrates how to implement some basic concepts of algorithmic music composition in the Java language. He presents code examples and resulting MIDI files generated by the Automatous Monk program, which uses the open source jMusic framework to compose music based on mathematical structures called cellular automata.

Many computer programmers are accomplished musicians; it's the rare software company that can't put together a decent house band. However, many programmer/musicians might be unaware of an interesting area in which their vocations and avocations intersect: algorithmic music composition. Algorithmic music composition is the application of a rigid, well-defined algorithm to the process of composing music. Cellular automata (CAs) -- a class of mathematical structures that evolve over time -- present an intriguing avenue for algorithmic music composition. Computers are ideal for computing the evolutions of a cellular automaton (CA) and displaying them graphically. You can also represent the evolutions with sound, including music. But finding techniques of mapping CA evolutions into pleasing and interesting music is a highly nontrivial problem. This article presents some techniques for doing CA-based musical composition in the Java language and explores specific mappings that yield especially good results.

An overview of cellular automata

A CA consists of:

- A matrix, or grid, of cells, each of which can be in one of a finite number of states
- A rule that defines how the cells' states are updated over time

The matrix of cells can have any number of dimensions. Given a cell's state and the state of its neighbors at time $t$, the rule determines the cell's state at time $t + 1$. (This will become clearer after you look at some concrete examples.)

Elementary cellular automata

I'll concentrate in this article on one-dimensional cellular automata, whose cells can be in one of two states, 0 or 1. Because the CA is one-dimensional, you can think of it as a row of cells. The
value of a cell at time $t + 1$ will depend only on the value of that cell and its immediate neighbors to the left and right at time $t$. Such CAs are called *elementary cellular automata*.

CA diagrams use white to represent 0 and black to represent 1. The top row shows the eight combinations of colors that a cell and its left and right neighbors can have. The bottom row shows the color of the center cell at the next step. For example, consider the fourth square from the left in Figure 1. In this square, you can see that if a cell is white, its left neighbor black, and its right neighbor white, then that cell will be black in the next step. The convention is to call this *rule 150*: If you think of the white and black cells as representing the binary digits 0 and 1, respectively, then the bottom row codes up the decimal number 150 in binary form. Figure 1 is a visual representation of rule 150. Rule 150 generates some interesting tunes when you use sound to represent a CA, so I'll use it as an example throughout this article.

**Figure 1. Rule 150**

Now, consider a rule 150 CA that starts out with all white cells, except for a single black cell in the center. The CA then evolves through the sequence of steps illustrated in Figure 2.

**Figure 2. Rule 150 sequence of steps**

Note that although the automaton is one-dimensional, you show sequential steps of its evolution in successive rows going down the page. Figure 2 shows the first five steps (including the initial state) of the CA's evolution. You can see that each cell's color is determined by its own color and its immediate neighbors' colors from the row above, according to rule 150. Also, note that you consider the values of all the cells in the row to be updated simultaneously at each step of the evolution.

Figure 3 shows how the CA looks after 100 steps of evolution:

**Figure 3. Rule 150 after 100 steps**

The CA evolution in Figure 3 happens to be symmetric, but not all CA evolutions are.

**Wolfram's research into cellular automata**

CAs have been a subject of research for over half a century. Stanislaw Ulam and John von Neumann invented the concept of CAs in the 1940s and made many important discoveries in the 1940s and 1950s. John Horton Conway and Bill Gosper did further studies in the 1970s on a particular two-dimensional type of CA that Conway invented, known as *Life*. Stephen Wolfram started doing CA research in the 1980s (see Related topics).
By studying elementary cellular automata, Wolfram found that complex behavior could arise from simple mechanisms. For example, consider rule 30. As with all elementary cellular automata, its definition, illustrated in Figure 4, is quite simple -- a small diagram defines it completely.

**Figure 4. Rule 30**

![Rule 30 diagram]

However, rule 30's subsequent evolution is quite complex. Figure 5 shows a CA's evolution after 100 steps, using rule 30.

**Figure 5. Rule 30 after 100 steps**

![Rule 30 evolution after 100 steps]

After examining the behavior of the 256 elementary CAs and other more complex CAs, Wolfram found that CAs could be categorized into four classes. Mathematician and author Rudy Rucker has described these four classes concisely and accurately in his presentation "Things Computer Science Tells Us About Philosophy" (see Related topics):

- **Class 1:** Constant. (Any seed "dies out.")
- **Class 2:** Repeats. (Loop, Stripes.)
  - **Class 2A:** Nested. (Regular Fractal.)
- **Class 3:** (Pseudo)Random. (Seething.)
- **Class 4:** Complex. ("Gnarly". Gliders. Universal computations.)

Wolfram has made the plausible claim that most class 3 and 4 CAs are probably *computationally irreducible*: Given an initial state, to find out the value of a particular cell at step $n$, you must run out the computation through all $n$ steps starting with the initial configuration. That is, there's no formula or shortcut that allows you to predict the CA's future state.

**Beyond music**

Can CAs' computational power be put to uses other than musical composition? Check out the sidebar "Applications of cellular automata."

**CAs' computational power**

Moreover, Wolfram, and Matthew Cook have proven that rule 110 is computationally equivalent to a universal Turing machine. (Conway proved this earlier for Life.) That is, you can use rule 110 to compute any function that a universal Turing machine can compute. This is probably true for other elementary CAs in class 4 as well. That is, some CAs, although they have simple definitions, can be programmed to perform any desired computation.

**An open problem**

Of course, CAs are pure mathematical constructs; visual representations of CAs simply help us understand and talk about them. In "Open Problems & Projects," Wolfram posed this problem: Is it possible to come up with an audio representation of CA evolution that provides insight that visual
representations don't provide? It's an interesting problem. You can examine an image at your leisure and linger over points of interest, but you experience a sound through time; when it's over, the best you can do is play it again. Wolfram thinks that for this and other reasons it's probably not possible to come up with such an audio representation.

A musical problem
I'll focus on a slightly related problem: Is it possible to come up with interesting and pleasing music using CAs? Pictorial representations of some CAs can be quite striking and beautiful. Can this beauty be rendered in music too? Also, it would be nice to harness CAs' universal computational power to compose music -- in particular, to find a way to map CA evolutions into music that's pleasing or at least interesting.

Using the Java language for musical composition
jMusic, created by Andrew Sorensen and Andrew Brown, is an open source framework for writing music programs in the Java language (see Related topics). It's built on top of the Java Sound API and allows composers to think at the musical level without having to worry about low-level audio programming details. Construction of a musical score in jMusic is elegant and intuitive. It mirrors the construction of musical scores on paper: Notes make up a phrase, phrases make up a part, and parts make up a score. The example in Listing 1 illustrates this simplicity.

Listing 1. "Bicycle Built for Two," by Harry Dacre, in jMusic
```java
int[] pitches = { C5, A4, F4, C4, D4, E4, F4, D4, F4, C4 };
double[] rhythmValues = {
    DOTTED_HALF_NOTE,
    DOTTED_HALF_NOTE,
    DOTTED_HALF_NOTE,
    DOTTED_HALF_NOTE,
    QUARTER_NOTE,
    QUARTER_NOTE,
    QUARTER_NOTE,
    HALF_NOTE,
    2 * DOTTED_HALF_NOTE
};
Note[] notes = new Note[pitches.length];
for (int i = 0; i < notes.length; i++) {
    // A note is made up of a pitch and duration
    notes[i] = new Note(pitches[i], rhythmValues[i]);
}
// A phrase is made up of notes
Phrase phrase = new Phrase(notes);
Part pianoPart = new Part("Piano", PIANO);
// A part is made up of phrases
pianoPart.add(phrase);
// A score is made up of part(s)
int tempo = 180;
Score daisy = new Score("A Bicycle Built For Two", tempo, pianoPart);
// In key of F: 1 flat
daisy.setKeySignature(-1);
// In 3/4 time
daisy.setNumerator(3);
daisy.setDenominator(4);
// Display score in standard musical notation
View.notate(daisy);
// Write out score to MIDI file
Write.midi(daisy, "C:\Daisy.mid");
```
Now, listen to the resulting MIDI file.

jMusic has many other nice features in addition to MIDI functionality. For example, you can display your score in conventional music notation. Also, the mod package lets you perform transformations on phrases, such as transposing or inverting.

**Automatous Monk**

Automatous Monk (a.k.a. Monk), named after the mighty jazz pianist and composer Thelonious Monk, is an open-source program I've written in the Java language that generates melodies from CA evolutions (see Related topics). Monk uses the jMusic framework to represent the resulting music as jMusic scores you can save and play as MIDI files. I'll use code examples from Monk and short MIDI file examples created by Monk to illustrate my points.

**Representing CAs in the Java language**

I won't go into detail on how to fully represent CAs in the Java language, but it's instructive to see how a cell's next state can be computed from its current state and its neighbors' current states. A binary number, as you've seen, can be used to represent a rule, and an integer can be used to represent the state of a CA cell at a given step. The code in Listing 2 calculates a cell's next state.

**Listing 2. Calculating the next state of a cell**

```java
/**
 * Computes the next state of a cell.
 * @param nWCellState State of the cell to the left of the current cell in preceding generation.
 * @param nCellState State of the current cell in preceding generation.
 * @param nECellState State of the cell to the right of the current cell in preceding generation.
 * @return Next state of the cell.
 */
int getNextStateOfCell(int nWCellState, int nCellState, int nECellState) {
    // Find the index of the current state in the rule pattern
    int index = 4 * nWCellState + 2 * nCellState + nECellState;
    // Get the value of the digit in that place
    int nextState = (rule >> index) % 2;
    return nextState;
}
```

**Musical representations of CA evolutions**

Now comes the hard part: How do you construct music from CA evolutions? One fact seems obvious: With visual representations of CA evolutions, time increases as you go down the page. This makes sense for a musical representation also. So, each row of the CA evolution will represent one beat.

I'll show you some mappings in Monk that give good musical results. Nested CAs tend to yield pleasing music; you can hear the visual regularity reflected in the music. The nested rule 150 is a good example. Wolfram's book *A New Kind of Science* briefly outlines some ways of representing CAs by music (see Related topics). My mappings are based on those ideas.
"Keyboard" mapping

This "keyboard" mapping is probably the most obvious one. Each cell (or column) in the CA corresponds to a particular pitch. You can think of each cell as mapping to a piano key. However, my implementation complicates the computation somewhat because it takes into account the type of scale you use -- for example, chromatic, major, or minor -- and the intervals between notes in that type of scale. The mapping in Listing 3 calculates the pitch for a given cell.

Listing 3. Calculating a cell's pitch

```java
/** *
 * @param cellPos Position of the cell. The position of the center cell is 0, while positions to the right are positive and positions to the left are negative.
 * @param normalize If true, make sure return value is a valid MIDI pitch. That is, make sure that it is between 0 and 128.
 * @return The starting pitch of the cell.
 */
int getStartingPitch(int cellPos, boolean normalize) {
    int[] scale = scaleType.getScale();
    int interval;
    int scaleWidth = scale.length;
    // Compute the interval from middle C to this pitch.
    if (cellPos >= 0) {
        interval = (cellPos / scaleWidth) * HALF_STEPS_IN_OCTAVE + scale[cellPos % scaleWidth];
    } else {
        interval = (-cellPos / scaleWidth) * HALF_STEPS_IN_OCTAVE;
        if (-cellPos % scaleWidth != 0) {
            interval += 12 - scale[scaleWidth - (-cellPos % scaleWidth)];
        }
        interval *= -1;
    }
    // Add interval to middle C, C4.
    int pitch = C4 + interval;
    if (normalize) {
        pitch %= 128;
        if (pitch < 0) {
            pitch += 128;
        }
    }
    return pitch;
}
```

Now, listen to the result when you apply this mapping to rule 150.

As you can hear, the result is a bit cacophonous (although you can hear its nested structure in a coarse way). The "chords" are simply too dense. In the rest of this article, I'll show you mappings that produce simple melodies without multiple simultaneous notes. These types of mappings, combined with the right rules, can generate surprisingly appealing and intriguing melodies.

"Row to binary number" mapping

The "row to binary number" mapping considers each row to be a binary number that represents a pitch. You encounter one subtlety, though: How do you order the digits? Usually, you order
them from right to left, with the right-most digit being the least significant digit. However, a one-dimensional CA has no right-most (or left-most) cell. We consider the CA's "universe" to be of infinite extent in both the left and right directions. So, as you do with the x-axis in the Cartesian plane, you consider a particular cell to be the origin. Cells to the right of the origin are numbered positively, and cells to the left of the origin are numbered negatively. All the example CAs in this article start with a single black cell, so it's natural to consider this cell as the origin.

In ordering the binary-number digits from least- to most-significant, start at the origin and work your way outward. You can do this in either of two ways, depending on whether you favor the cells to the left of the origin or the cells to the right of the origin:

- Favor left: 0, -1, 1, -2, 2, -3, 3, ...
- Favor right: 0, 1, -1, 2, -2, 3, -3, ...

In either case, you can code up a row of the CA as a binary number. Because only numbers between 0 and 128 specify MIDI pitches, you can then convert this number to a MIDI pitch by taking its value modulo 128. Listing 4 contains the corresponding code from Automatous Monk.

**Listing 4. Reordering a row and calculating the pitch**

```java
/**
 * Reorders a row according to significance of digits.
 * @param rowIndex The generation number of the row
 * @param cA The CA containing the row
 * @param bias Left- or right-side bias
 * @return The reordered row with least-significant digits to right
 */
int[] reorderRow(  
    int rowIndex,  
    CellularAutomaton cA,  
    HorizontalBias bias) {  
    int[][] cAHistory = cA.getGenerations();  
    int[] row = cAHistory[rowIndex];  
    int len = row.length;  
    int[] reordered = new int[len];  
    int mid = len / 2;  
    reordered[len - 1] = row[mid];  
    for (int i = 0; i < (len - 1) / 2; i++) {  
        // Note that this favors one side of the CA over the  
        // other side, but there seems to be no way to get around this.  
        if (bias.equals(HorizontalBias.LEFT)) {  
            reordered[len - (2 * i + 1) - 1] = row[mid - (i + 1)];  
            reordered[len - (2 * i + 2) - 1] = row[mid + i + 1];  
        } else {  
            reordered[len - (2 * i + 1) - 1] = row[mid + i + 1];  
            reordered[len - (2 * i + 2) - 1] = row[mid - (i + 1)];  
        }  
    }  
    return reordered;  
}

/**
 * Computes a MIDI pitch from a row of 0s and 1s.
 * @param rowIndex The generation number of the row
 * @param clllrAutmtn The CA containing the row
 * @param bias Left- or right-side bias
 */
```
* @return A valid MIDI pitch
*/
int getPitchFromRow(
    int rowIndex,
    InfiniteCellularAutomaton clllrAutmtn,
    HorizontalBias bias) {
    int[] reorderedRow = reorderRow(rowIndex, clllrAutmtn, bias);
    int pitch = 0;
    for (int i = 0; i < reorderedRow.length; i++) {
        pitch = (2 * pitch + reorderedRow[i]) % 128;
    }
    return pitch;
}

Now, listen to the melody generated by applying this mapping to rule 150. Note that you get the same result whether you use left- or right-bias, because rule 150 is symmetric.

"Cumulative row to binary number" mapping

The "cumulative row to binary number" mapping is the same as the previous mapping, but you keep a cumulative sum -- a running total -- of the row values. The mapping code is in Listing 5.

**Listing 5. Keeping a running total of pitches of successive rows**

```java
/**
 * Converts the successive rows of a CA to a musical phrase.
 * @param cA A cellular automaton
 * @param bias Favor left- or right-side in generating pitches
 * @return A musical phrase corresponding to cA
 */
public Phrase convertCAHistoryToPhrase(
    CellularAutomaton cA,
    HorizontalBias bias) {
    Phrase phr = new Phrase();
    int cumulativePitch = 0;
    for (int i = 0; i < cA.getGenerationCnt(); i++) {
        int pitch = getPitchFromRow(i, cA, bias);
        // Need to take mod 128 to keep it a valid MIDI pitch.
        cumulativePitch = (cumulativePitch + pitch) % 128;

        Note n = new Note(cumulativePitch, CAConstants.DEFAULT_NOTE);
        phr.add(n);
    }
    return phr;
}
```

Now, listen to what that sounds like.

This result is more melodic than the previous example, which sounded a bit like a rhythm section.

**Combinations of mappings**

Combining mappings can lead to good results. For example, you can combine the rule 150 melody with the rhythm section. You can also combine the binary and cumulative binary mappings applied
to rule 60. Note that the left- and right-bias versions of the mappings are different, because rule 60 isn't symmetric. This leads to four voices and a richer sound.

I have to admit that I think rule 60 is rather catchy! I've spent a lot of time (certainly more than most people) listening to CAs, and I often find rule 60 playing in my head. I think of it as sort of the Hanson or the Jackson 5 of the CA world.

How to convert MIDI files to cell phone rings

It would be neat to generate cell phone rings with CA. (Wolfram has mentioned that he'd like such a ring for his own phone.) I'm not an expert on the subject, but cell phone ring mechanisms seem to vary across cell phone brands. However, at least a couple of open source software programs -- Clint Levijoki's midi2rtx and Andreas Leitner's Midi2C25 -- transform MIDI files into cell phone rings (see Related topics). I haven't used these programs yet, because I don't have a cell phone -- but I'm tempted to get one just to try them out.

Keep it simple

As I mentioned earlier, I've found it's best to concentrate on mappings that produce single-voice melodies (while perhaps layering these melodies as I did with the binary and cumulative binary mappings), and essentially ignore harmony and rhythm. This isn't as big a limitation as it might seem to be. Much baroque music, such as J.S. Bach's, was of this type. There's even a word for it: polyphony. (Of course, I'm simplifying things quite a bit: In polyphony, harmonies do emerge from the separate voices, and rhythm isn't completely ignored.) These polyphonic combinations of mappings are a good first step in creating music from CAs.

I've also found that simple mappings yield the most pleasing results. However, this shouldn't be surprising, because even simple visual mappings -- map a 1 to a black square and a 0 to a white square -- can generate quite beautiful and intricate patterns. Moreover, consider the fact that some CAs, such as rule 110, are capable of universal computation. And assume for the moment that musical composition is a purely rational activity -- that you can program a computer to compose music. Then it should be possible for a universal CA, such as rule 110, to generate music on its own without our having to "code up" any musical intelligence or knowledge in the mapping itself. Of course, I've been using only extremely simple starting states, and perhaps rule 110, say, needs a more complicated program and input data coded up in its starting state to produce truly beautiful music. The point is that the mappings from CA states to notes should be fairly "transparent;" the CA should do all the compositional work.

Another danger in trying to make your mappings complex is that you tend to code up your own musical prejudices (be they Eastern or Western, tonal or atonal) in those mappings. Writing a computer program that can compose a piece in Chopin's or Bach's style nearly as well as Chopin or Bach could is a difficult problem and an admirable achievement; after all, it's always nice to have another "new" Chopin or Bach piece. But why limit ourselves to this? I think algorithmic music composition won't have truly succeeded until it can produce not only beautiful music, but also beautiful music of a wholly different type from what we've heard before. Chopin and Bach weren't great simply because they wrote great music; they were great because they introduced whole new musical ways of thinking. I think the best results will come from using simple mappings and letting the CA generate any musical ideas.
The Gnarl

Rudy Rucker has defined a concept he calls the *Gnarl*: things that have a level of chaos that is tuned right to the boundary between order and disorder.

When I first started listening to the CA music produced by Monk, some of the best-sounding CAs seemed to me to be examples of the Gnarl. Later, I read in Rucker's "Things Computer Science Tells Us About Philosophy" that he considers class 4 CAs to be examples of the Gnarl.

What does a gnarly CA sound like? Listen to rule 225.

When I listen to rule 225, I feel that I can almost grasp its structure, but not quite. This makes it possible to listen to it for long periods of time without getting bored. On the other hand, rule 60, which is a nested pattern, is quite enjoyable to listen to from time to time but doesn't hold my interest for long. Rule 60's structure is too easy to grasp; it might be nice, chewy, crunchy mind-candy, but it's not an example of the Gnarl. Even though rule 225 yields a regular nested pattern with the starting state I've given it -- and is therefore not in class 4 -- it still sounds pretty gnarly to me. (Why a CA can often sound more gnarly than it looks is a topic for a whole other article.)

So what does a real class 4 CA sound like? Rule 110 is in class 4; listen to it.

It definitely does sound as if we've entered the Lair of the Gnarl. As with much good music, you need to listen to it a few times before you can appreciate it fully.

Other algorithmic music composition projects

A lot more research is going on in algorithmic music composition than you might expect. There are basically two approaches to using computers for music composition: top-down and bottom-up. Composers who use computer programs as aids to composition mostly use them in a top-down manner -- by arranging musical building blocks into larger structures. Automatous Monk takes a bottom-up approach: It creates music at the note level.

CAMUS

CAMUS, written by Eduardo Reck Miranda, is another bottom-up approach that uses CAs to generate music (see Related topics). It uses Life and Demon Cyclic Space to generate music. Unlike the one-dimensional CAs I've shown you, both Life and Demon Cyclic Space are two-dimensional CAs; that is, the cells are arranged in a two-dimensional grid. Life is especially
interesting (and fun to program and watch). CAMUS uses Life to generate groups of notes while using Demon Cyclic Space to generate the timbre (or texture) of the individual notes.

**Music Sketcher**

Music Sketcher is an IBM alphaWorks project (see [Related topics](#)). Unlike Monk and CAMUS, Music Sketcher takes a top-down approach in that it lets the user arrange phrases or riffs into larger musical structures. The especially interesting thing about Music Sketcher is that it lets you create chord progressions for pieces while the program does the transformations necessary to ensure that the building blocks conform to the specified chord progression. (In this way, it's sort of a precursor to Apple's new GarageBand application.) It's a fun program, and I encourage you to try it out.

**Conclusion**

Many interesting approaches to algorithmic music composition using CA remain to be explored. The examples I've provided use only one simple starting state. What types of music can be found by "seeding" the CA with more complex starting states? Also, most Western pieces of music have chord progressions to which the melody conforms. It would be interesting to devise mapping techniques that cause CAs to generate melodies that conform to user-specified chord changes (a CA Charlie Parker, in other words), although that might limit CAs by making them conform to musical prejudices.

A limitation of the current version of Monk is that it only uses conventional Western pitches, the 128 MIDI tones. When you interpret a CA row as a binary number mod 128, you're effectively only looking at 7 digits or cells of the row. Seven cells is an awfully thin vertical strip of the complete CA; a lot of information is lost. We could preserve this information by using microtones, rather than only the semitones used in Western music. Would this lead to richer-sounding music or to mere discord? I plan to expand Monk to try some of these ideas.

Automatous Monk and jMusic are both freely available on SourceForge.net, and I encourage you to try your own experiments. You might also want to take a look at Wolfram's "Open Problems & Projects" paper for ideas. Approaches to the science of cellular automata can come from many angles and involve varying degrees of technical sophistication, from high school level to graduate level. In the words of Rudy Rucker, "Seek ye the Gnarl!"
Related topics

- Watch IBM Bluemix in action: In this demo, David Barnes shows you how to develop, create, and deploy an application in the cloud.
- An excellent work on cellular automata (and much more) is Stephen Wolfram's book *A New Kind of Science* (Wolfram Media, 2001). Also, take a look at Wolfram's article "Open Problems & Projects" for fun things to work on. All CA illustrations in this article are from *A New Kind of Science*.
- If you want to read more about Wolfram, von Neumann, and some of the other interesting people who worked at Princeton University's Institute for Advanced Study, read Ed Regis's entertaining book *Who Got Einstein's Office?* (Addison-Wesley, 1987).
- Mathematician and author Rudy Rucker, a major influence on this article, has many interesting and valuable ideas on CAs and other math topics. Check out these works in particular:
  - *A New Kind of Science At San Jose State University*
  - **Automatous Monk** is an ongoing open source project. Download the source code this article uses. You can also hear extended versions of the elementary CA rules at the Monk Web site.
  - Download Music Sketcher from alphaWorks.
  - For more mathematical music, listen to the pieces on Daniel Cummerow's The Sound of Mathematics Web site.
  - Find out more about CAMUS.
  - Learn more about Life.
- Check out the midi2rtx and Midi2C25 projects for converting MIDI files to cell phone rings.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)