

Musical Programming Languages: A Survey

Sigmon Myers

University of California Santa Cruz
Computer Science Department
sig@soe.ucsc.edu

Abstract

Musical programming languages haven't historically been an area of immense interest within the programming languages community. After conducting a brief survey of this niche research area, we argue that musical programming languages are able to benefit the computer science community not only as an artistic, domain-specific endeavor, but also as a means of introducing and educating students relatively new to the field.

We present previous works, which led to various new programming paradigms such as "On-The-Fly Programming" and discuss how and why these paradigms might be useful in various ways to programming language research.

General Terms Music, Programming Languages, Computer Science Education.

Keywords On-The-Fly Programming, Object-Oriented Music, Aspect-Oriented Music.

1. Introduction

Musical programming languages are generally not a widely discussed topic in the programming languages community at large. This paper intends to present a survey of various music and music-related programming languages and their respective paradigms in order to understand how and why we might find these languages useful in the research community.

Over the last few decades we have seen music transition from being generated solely in hardware (the only time-sensitive means of delivering sound to an audience) to having its roots set in software. This transition allowed us to take more advantage of sound manipulation by means of modifying the underlying algorithms used to produce various sounds. Musical programming languages aren't solely used as an abstract form of art—they can be used to create everything from classical music to funk, from avant-garde to pop. This distinction is necessary as we present material, which can be used across a multitude of musical genres.

It is also our intention to provide a means of understanding how musical programming languages could be used in an educational setting in order to introduce students to the computer science field who, previously, would have had little or no exposure (or interest!).

Our paper continues by presenting a previous work in section 2. Section 3 presents relatively recent research done with aspect oriented programming and music. Section 4 presents a discussion of On-The-Fly Programming and the various artistic implications of this interesting new paradigm. In Section 5 we draw our conclusions based on the works presented as a means of arguing for more work to be done in this unique field.

2. Previous Works

To fully understand where musical programming languages are going in the future, it is worthwhile to consider where they have come from in the first place. We could start from the first electronic synthesizer, the Musical Telegraph, developed by Elisha Gray in 1876 [2]. In the interest of conciseness we would like to instead present the work of Scaletti and Johnson who pioneered the use of object-oriented music composition and created an interactive environment for manipulating such objects [1].

Sound synthesis was a costly endeavor at the time of this paper's publication in 1988. To synthesize a frequency, over 1 million 16-bit samples would be required to produce less than one minute of sound—hardly manageable for the computers of that era! The goal of Scaletti and Johnson in using musical "objects" was to handle generation of a sound object and synthesis of said object in as timely a manner as possible.

2.1 What about Music Notation?

In order to understand why sound objects were necessary for Scaletti and Johnson, we must mention standard musical notation. Figure 1 shows an example excerpt from a popular musical score, the main theme from Star Wars. This serves as an excellent example of music to be interpreted (compiled?) and performed (executed?) by a musician. Music, however, cannot be solely defined by music notation. There are numerous examples of music, which cannot be accurately transcribed by a pen and paper. In order to get a grasp of this idea, we encourage you to look into Herbie Hancock's "Headhunters" album that makes use of many synthesized sounds that standard music notation is unable to notate.

2.2 Sound Objects

Drawing from the ideals of object oriented programming, everything is inherited from a sound object in this particular paradigm. Scaletti defines sound objects quite well: "some means is needed by which to organize these samples, group them together into meaningful chunks, enclose all the details in a package, give it a name and refer to it thereafter as a single entity."

A single sound object on its own is rather useless. A set of



Figure 1. Standard musical notation is structured and has semantics, quite similar to programming languages, however musical notation is not necessarily able to encompass all music.

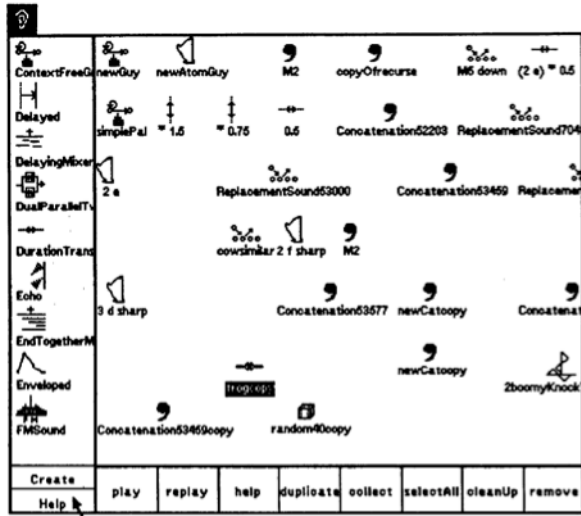


Figure 2. Standard musical notation is structured and has semantics, quite similar to programming languages, however musical notation is not necessarily able to encompass all music.

sound objects is therefore used to create a musical composition. Assuming we have some set of sound objects, we might want some means of manipulating those objects so that we can further influence a composition. For instance, say we wanted to change our theme from Star Wars in Figure 1 to be three musical pitches higher. To do this we need to make use of N-ary (i.e. multiple) or Unary (i.e. single) Transformation on the sound object. Transformations have the capability of not only changing pitches of sound objects, but delaying, repeating, creating a palindrome of the sound, etc. Therefore, a transformation is merely the result of applying some function to the set of sounds, which is to be modified.

Once we have a set of sound objects and we've done our transformations, it is time to *synthesize* our musical composition. Synthesis comes from the *Lookup* function resulting in a sound being generated into a waveform (which can then be played back to the user).

To further discuss the notion of the *sound object* we introduce a comparison of an object-oriented language, Java [3]. Java inherits everything from a base object, intuitively called *Object*. Primitives in Java consist of *int*, *double*, and *bool*. Similarly, in object-oriented music, every object is a subclass of the *SoundObject* and our primitive in this case consists of the *SoundAtom*. Java supports the ability to have concurrent threads, methods are capable of doing specific pieces of functionality (such as sorting), and inheritance is implied in program development. Similarly, object-oriented music allows for multiple sound objects to be played concurrently and inheritance is also a key method for musicians to develop new and unique sound objects.

With this brief introduction to sound objects and object-oriented music, we present the authors' implementation of Kyma, which is a vehicle for creating/manipulating sound via a graphical user interface.

2.3 Kyma

Kyma was created in Smalltalk-80 in order to allow a musician without computer science expertise to be able to create, manipulate and compose on a computer. Figure 2 shows the Kyma interface. The left side of the image represents classes that can be called to create sound instances. Instances of sound objects appear to the right and can be played, manipulated, or further expanded using the methods, which appear along the bottom of the

image. The few methods of interest allow for sound objects to be played, replayed, duplicated, collected or removed.

This interface can be used to create a collection of sound objects, which can be extremely complex (i.e. a complete orchestration) or very fine grained (i.e. a clapping noise). Once a composer has defined all the attributes for a set of sound objects, he or she can generate a waveform to listen to the performance by choosing the *Play* method which in turn calls the *Lookup* function. Once this method is invoked, a directed acyclic graph is created which pieces together how the sound object should 'sound.' Figure 3 shows us an example of the graph created by a *Mixer* method call. We can see that three distinct sounds are being created (each sound being a child of the *Mixer*). The first sound consists of a *Frequency Transformation* (change of pitch), the middle child incurs a *Delay* (a pause for some period of time) and then does a *Frequency Transformation*, and the third child incurs a *Delay*, *Frequency Transformation* and then a *Duration Transformation* (meaning the sound is held longer). These three sounds are created concurrently (consider a real-world mixer which plays multiple sounds simultaneously) and result in a waveform being generated once the *Lookup* function is reached.

2.4 Analysis

The creators of Kyma envisioned this project to allow for musicians to create music instantaneously without being inhibited by strict algorithms existing in hardware synthesizers. They also envisioned using the object-oriented paradigm as a means to reduce overhead and increase the speed by which a sound can be played back to the end user. As one can imagine, in the 1980's, algorithms implemented in software for sound synthesis were incredibly slow to produce a result—often taking hours or days for small segments of a composition. Creativity, the authors argued, was inhibited by this restraint. In creating this system, the authors managed to reduce this wait, often to a matter of minutes, which arguably allows for a less inhibited creative process to occur.

The authors eventually formed a business around Kyma, which

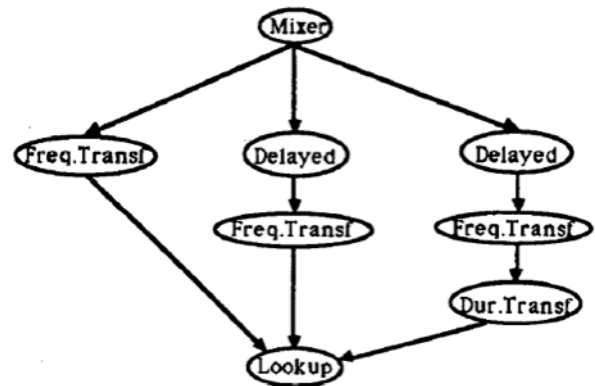


Figure 10. A Simple N-ary Transform Structure in Kyma.

Figure 3. Standard musical notation is structured and has semantics, quite similar to programming languages, however musical notation is not necessarily able to encompass all music.

is currently in its tenth iteration and named accordingly, Kyma X. The authors maintained their interest in having the end user be able to freely create and manipulate sound objects as much as possible by incorporating all of the same transformations and

capabilities that existed in their original research version. Figure 4 shows a screenshot of the current Kyma X interface. Kyma X is currently able to serve as a multi-track recorder, allowing for input from an external audio source, as well as placement of sound objects (light-blue rectangles) at specific time intervals in an audio track.

3. Aspect Oriented Music

As programming languages evolved, so did the domain of computer music. Though computer music is often tangential to programming languages research at best, it obviously relies heavily on developments produced by the programming languages community. Hill, Holland and Laney did work with aspect-oriented programming which allowed for music, in this instance, to contribute to the programming languages community. It is worthwhile to first discuss AOP in its general form before we discuss how Hill et al. incorporated it into a domain specific context.

In general, Aspect Oriented Programming (AOP) [10] is a means of addressing crosscutting concerns in a program. AOP utilizes three techniques: Advice, Pointcuts, and Aspects. A pointcut is a spot in a program where a programmer wants some piece of advice to be applied. Advice is the set of code to execute at a given pointcut. An aspect is the combination of a pointcut injected some set of advice to address a given crosscutting concern. Historically, AOP has been utilized well for logging functionality, but is often difficult to implement outside of that context.

3.1 Symmetric Composition of Musical Concerns

Hill et al. managed to take AOP and extend it to the point where it could be applied to another domain altogether (and prove useful!) [4] [5]. They argue that this is the first implementation of AOP in a domain-specific setting of which AOP researchers argue many domain specific contexts exist.

To understand how aspects are useful in the musical context we offer a brief explanation. Western music consists of rhythms and melodies. A musical score can consist of thousands of rhythms and notes (which makeup melodies), and a score can be further subdivided until a fine granularity of a score is achieved. Imagine if pieces of that score could be changed according to some piece of *advice*, or better yet, be altered according to several *hundred* pieces of advice. We might be able to take an original score and transform it so much with aspects that, when played, we cannot even recognize the original!

Aspects in music are well aligned to allow musicians to ex-

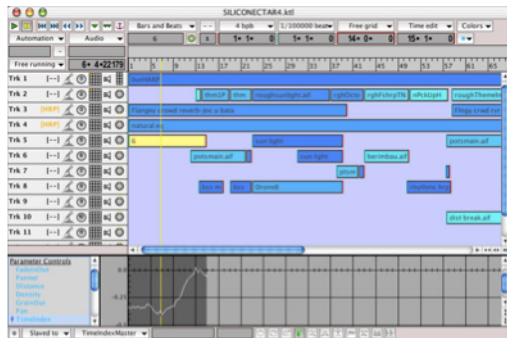


Figure 4. Kyma X has evolved from to a consumer product and includes all functionally present in the original research prototype.

plore rapid “what-if” scenarios, where the musician applies an aspect to see if the alteration made a musical impact or not. This

CMU1

Type	Sequence				
#pitch	I	1	2	3	4
S	<C>	<D>	<E>	<F>	

CMU2

Type	Sequence				
#rhythm	I	1	2	3	4
S	<1>	<2>	<2>	<1>	

CMU1 and CMU2 composed in sequence

Type	Sequence				
#pitch	I	1	2	3	4
S	<C>	<D>	<E>	<F>	
#rhythm	I	1	2	3	4
S	<1>	<2>	<2>	<1>	

Figure 5. CMU1 is a sequence of pitches; CMU2 is a sequence of rhythms. CMU1 and CMU2 could be composed in sequence to result in the third item in this figure.

is largely a non-scientific process. The distinction in this domain must be made between a computer scientist (or any scientist!) whose intent is to problem solve whereas a composer or artist intends to problem seek.

Aspects can be applied to musical scores by means of modifying existing rhythmic and melodic patterns. Rhythm refers to the beat or pulse of a musical idea whereas the melody consists of intervals between various notes (i.e. three distinct notes, C, E & G would be apart of a melody whereas *how* those notes are played would define the rhythm).

3.2 How does Aspect Oriented Music work?

In Aspect Oriented Music, we find that instead of a sound object, we have *Music Units*. *Music Units* are all completely disjointed from each other unless they are put together into a container, called a *Composed Music Unit*. *Composed Music Units* can also, subsequently, contain other *Composed Music Units*. Importing a musical score from a format such as MusicXML or MIDI allows us to create our *Music Units*. MusicXML is an XML-based representation of a musical score, whereas MIDI is a standard for specifying the duration and pitch of a given note. Once a composer has a score to manipulate, they can specify music units, composed music units and much more.

Figure 5 shows us an example of several *Composed Music Units*. CMU1 consists of the musical pitches C, D, E and F. CMU2 describes a rhythmic sequence consisting of 1 beat, 2 beats, 2 beats, and then 1 beat. If we wanted to “line up” these two sequences together, we could do so by employing a tactic in aspect oriented programming known as *weaving*. Weaving is the method of injecting the necessary code into the executable to ensure that wherever a given aspect occurs, the code is generated (via a transformation) in the correct place by the compiler. For more information, we again refer you to [10]. The end result of weaving the two *CMU’s* in sequence would be the third item in Figure 5. We can see that our pitches C, D, E, and F, are aligned properly with our sequence of beats. This sort of sequence can only occur on an ordered collection.

3.3 Really? Is that it?

There’s more--we can compose *Music Units* in a variety of ways! We could take two *CMU’s* of the same type (pitch, for instance) and compose them in parallel resulting in chords, which are essen-

CMU1				
Type	Sequence			
#pitch	I	1	2	3
	S	.C	.E	.G

CMU2				
Type	Sequence			
#pitch	I	1	2	3
	S	.E	.G	.B

CMU1 and CMU2 composed in parallel				
Type	Sequence			
#pitch	I	1	2	3
	S	.C.E	.E.G	.G.B

Figure 6. Similar to Figure 5, we see that performing a parallel weaving will result in CMU1 and CMU2 being composed together to create chords—or a set of notes, which play in unison.

tially multiple notes played simultaneously. Figure 6 displays this idea by composing two CMU’s in parallel.

Now that we know what these various units are and how they can be transformed, let’s look at a larger example. When dealing with multiple CMU’s for which we want to inject advice, we consider the set of CMU’s to be a hyperspace. Each row in our hyperspace is a hyperslice (a CMU put into a dimension). Figure 7 shows us an example hyperspace in which we have two phrases, *Phrase1* and *Phrase2*. Each CMU is representative of a different concern within a given phrase, which is why we see three *Phrase1* dimensions, for example.

In our hyperspace we can compose various different melodies depending on how we apply our advice. Figure 8 depicts an example aspect, which will tell us how to create a short composition from our hyperspace. In parsing this bit of code we see that we are taking two hyperslices, *Phrase1.** and *Phrase2.** and creating a composition with the units A, B and R (the *overrideByName* relationship merely tells us to take the last hyperslice given two or more matching hyperslices to a given piece of advice—in this case the fourth row is applied as opposed to the third row from the hyperspace). Our resulting composition is displayed in Figure 9—a true masterpiece!

3.4 Analysis

The authors of this paper applied their methodology to a series of J.S. Bach’s scores in MusicXML format. Bach’s music, in particular, provides an excellent platform for creatively modifying the composer’s works into something completely unique. The possibility of applying aspects to music doesn’t only have merits in relation to classical music, however. This author believes that video games would make especially good use of aspect-oriented music. Imagine a series of video game levels, which continually became more and more cryptic in nature. The music could have transformations applied at specified intervals which—apart from being completely unique each time—would have the capability to



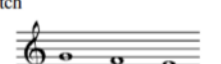

Dimension	Concern	Unit	Unit Implementation
Phrase1	Melody	A	#pitch 
Phrase1	Rhythm	R	#rhythm 
Phrase1	Melody	B	#pitch 
Phrase2	Melody	B	#pitch 

Figure 7. A Hyperspace consists of multiple CMU’s—each row (CMU) is considered to be a Hyperslice.

```
Musicunit: Intro2
Hyperslices: Phrase1.*, Phrase2.*
Relationships:
    overrideByName;
Composition:
    ".*A" + ".*B" + ".*R";
```

Figure 8. An aspect’s advice is applied to *Phrase1* and *Phrase2* (see Figure 7) resulting in a transformation (see Figure 9).



Figure 9. The transformed composition after applying the advice shown in Figure 8.

become more and more dark by specifying melodies known by composers to invoke a certain emotion from a person.

While no video games or other areas are incorporating aspect-oriented music in a consumer-level product, it is arguable that music on memory or space constrained resources could (possibly significantly) be reduced by introducing aspects on pre-determined excerpts of music. This would reduce space and might be an ideal implementation for casual video games not requiring cinematic-quality graphics and audio.

4. On-The-Fly Programming

On-The-Fly Programming is one of the most unique forms of musicianship in existence today. The programming paradigm, also referred to as live coding, is the practice of creating music (often in front of an audience) by means of computer software. There are three well-known tools today that make use of the live-coding paradigm: Impromptu [6], Chuck [7], and Supercollider [8].

This paradigm consists of a programmer writing a segment of code that is continually debugged and executed by one of the previously mentioned tools. The result of which, is sound being continually produced, manipulated, refined and expanded upon. In recent years, laptop orchestras have started to be created (mainly at universities such as Stanford and Princeton) where the code to be produced is displayed on a backdrop accompanied with visual effects as the music is being produced. Ultimately, this paradigm differs from most any other programming paradigm in its focus on programmer creativity as opposed to software engineering requirements. Figure 10 depicts a typical laptop orchestra performance where the practice of live coding occurs.

In general, on-the-fly programming languages share some unique characteristics as compared to other programming languages. First off, the programming languages explicit knowledge of time makes it possible for a live code to develop and deploy music in real-time and have it turn out as expected. Without this capability, it would be extremely difficult to generate sound on-the-fly with any sense of how and where different musical ideas (code fragments) should intertwine with each other.

Another unique characteristic of on-the-fly programming is that the source code is generally compiled and output continually—after a programmer inputs a value for a musical chord structure, the audience will start to hear that particular chord structure. A slight change to one of the chords in the set could evoke a different emotion in the audience. This aspect on live coding is synonymous with creativity and composition on the programmer’s part.

Thirdly, this paradigm offers insight into other ways in which we, as computer scientists, can make use of this model. Interactivity can easily be lacking in a computer science classroom. Incorporating live-coding techniques based on a template could encourage exploration and serve as a teaching aid in the class-



Figure 10. A live coding performance where source code and visual effects are displayed on projectors while music is being generated by software.

room. For those students not interested in music, there are also a plethora of art-generation languages that could be utilized as well. As live coding progresses out of its infancy it will be interesting to see how it is adapted in both education and software engineering settings.

5. Conclusions

To wrap up our short survey on various music-related programming languages, it is necessary to discuss Blackwell and Collins' "The Programming Language as a Musical Instrument" [9]. They argue for more exploration in the domain of live-coding and, in particular, its application to software engineering. As computer scientists, we often build programming languages that are convenient for us (engineers) to use. JavaScript, at its onset, was a programming language that was easy to use for a large number of non-engineers to use—this clearly became a pivotal force in shaping the Internet. What if we start shaping all of our programming languages around general-users with the idea that they might take our programming language in a completely different direction than we imagined? The ramifications of this can be difficult to manage (as demonstrated by the JavaScript standardization team)—but these are *good* problems to have.

Blackwell and Collins also argue for studying unusual programming populations as a means of developing new programming languages and paradigms. There is merit to this argument—we often refer to this practice as developing domain specific languages (DSL), but designing programming languages, which experts in their own field of study can utilize increases knowledge about computer science in addition to understanding how an end-user intends to interact with a system. Some programming languages originally developed as a DSL's turned out to be rather decent general-purpose programming languages. The authors cite Smalltalk (originally developed for children) and the spreadsheet (originally developed for business students) as examples of this phenomenon.

In closing, the arguments exist for pursuing musical programming language research with the hope of furthering both the computer science and music fields. As we delve deeper and deeper into different programming paradigms, software engineering methodologies, and programming language creation it is worthwhile to consider Nobel's viewpoints on Postmodern Programming. Nobel argues that computer scientists will pursue programming as an activity, which allows for creativity, performance, and an undefinable/abstract end product. All in all, that sounds a lot like music to me!

6. Acknowledgments

Thanks to Dr. Van Gelder for making this class possible this quarter, as well as to Caitlin for taking the time to teach it!

References

- [1] C. A. Scaletti and R.E. Johnson. *An Interactive Environment for Object-oriented Music Composition and Sound Synthesis*. Proceedings of the 1988 Conference on Object-Oriented Programming Languages and Systems, pp. 18-26, ACM, 1988.
- [2] Sweezey, Kenneth M. 120 Years of Electronic Music. <http://120years.net/machines/telegraph/index.html>
- [3] Java programming language. <http://java.sun.com/>
- [4] Hill, P., S. Holland, and R. Laney. 2006. *Symmetric Composition of Musical Concerns*. Proceedings of the 5th International Conference on Aspect-Oriented Programming (AOSD '06). New York, New York: Association for Computing Machinery, pp. 226-236.
- [5] Hill, P., Holland, S. and Laney, R. *Using Dynamic Aspects in Music Composition*. Dynamic Aspects Workshop, AOSD '04 International Conference on Aspect-Oriented Software Development, Lancaster UK, 2004, 89-97.
- [6] Sorensen, A. 2005. *Impromptu: an interactive programming environment for composition and performance*. In Proceedings of the Australasian Computer Music Conference, 2005.
- [7] Wang, G. and Cook, P.R. 2003. *ChucK: A Concurrent, On-the-fly, Audio Programming Language*. Proceedings of the International Computer Music Conference. ICMA, pp. 219-226.
- [8] McCartney, J. 1996. *SuperCollider: A New Real-Time Synthesis Language*. In Proceedings of the International Computer Music Conference. International Computer Music Association, pp. 257-258.
- [9] Blackwell, A. and Collins, N. 2005. *The Programming Language as a Musical Instrument*. In P. Romero, J. Good, E. Acosta Chaparro and S. Bryant (eds.) Proceedings of the 17th Workshop on the Psychology of Programming Interest Group. Sussex University, pp. 120-130.
- [10] Laddad, Ramnivas. *I want my AOP!* JavaWorld: Solutions for Java Developers. <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>