Defining Programming languages for Music through the view of “Somewhat Weak” Computer Music

Anonymized for review

example@example.com

ABSTRACT

This paper critically reviews the history of programming languages for music by referring discussions from sound studies, aiming to describe this history decoupled from computer music as a form/community. This paper focuses on critiquing the discourse of Post-Acousmatic, which inclusively addresses recent trends in computer music. The universalism associated with pulse-code modulation (PCM), which is basic assumption of today’s sound programming, has functioned as a discourse that shapes musicians’ expectations historically, despite the fact that its expressive range has several limits. Also, this paper points out that the MUSIC-N family, which formed the foundation of PCM-based sound synthesis, is contextualized not as programming languages in terms of their syntactic or semantic properties, but as a lineage of workflows for generating sound on computers, and these systems have evolved into black boxes that minimize the need for users to understand their internal structures over time. The paper concludes that programming languages for music developed since the 2000s function as a means of presenting alternatives to the often-invisible technological infrastructures surrounding music, such as formats and protocols, rather than solely aiming to create novel musical styles. This conclusion paves the way for future discussions in this research area.

1. INTRODUCTION

Programming languages and environments for music have developed hand in hand with the history of creating music using computers. Software and systems like Max, Pure Data, CSound, and SuperCollider has been referred to as “Computer Music Language” [1, 2, 3], “Language for Computer Music” [4], and “Computer Music Programming Systems” [5], though there is no clear consensus on the use of these terms. However, as the shared term “Computer Music” implies, these programming languages are deeply intertwined with the history of technology-driven music, which developed under the premise that “almost any sound can be produced” [6] through the use of computers.

In the early days, when computers were confined to research laboratories and neither displays nor mouse existed, creating sound or music with computers was inevitably equal to the work of programming. Today, however, programming as a means to produce sound on a computer—rather than employing Digital Audio Workstation (DAW) software like Pro Tools is not usual. In other words, programming languages for music developed after the proliferation of personal computers are the softwares that intentionally chose programming (whether textual or graphical) as their frontend for making sound.

Since the 1990s, the theoretical development of programming languages and the various constraints required for real-time audio processing have significantly increased the specialized knowledge necessary for developing programming languages for music today. Furthermore, some languages developed after the 2000s are not necessarily aimed at pursuing new forms of musical expression. It seems that there is still no unified perspective on how the value of such languages should be evaluated.

In this paper, a critical historical review is conducted by deriving discussions from sound studies alongside existing surveys, aiming to consider programming languages for music independently from computer music as the specific genre. ### Use of the Term “Computer Music”

The term “Computer Music,” despite its literal and potential broad meaning, has been noted as being used within a narrowly defined framework tied to specific styles or communities, as represented in Ostartag’s *Why Computer Music Sucks* [7] since the 1990s.

As Lyon observed nearly two decades ago, it is now nearly impossible to imagine a situation in which computers are not involved at any stage from production to experience of music [8, p1]. The necessity of using the term “Computer Music” to describe academic contexts, particularly those centered around the ICMC, has consequently diminished.

Holbrook and Rudi continued Lyon’s discussion by proposing the use of frameworks like Post-Acousmatic [9] to redefine “Computer Music.” Their approach incorporates the tradition of pre-computer experimental/electronic music, situating it as part of the broader continuum of technology-based or technology-driven music [10].

While the strict definition of the Post-Acousmatic music is not given deliberately, one of its elements contains the expansion of music production from institutional settings to individuals and the use of the technology were diversified [9, p113]. However, while the Post-Acousmatic discourse integrates the historical fact that declining computer costs and access beyond laboratories have enabled diverse musical expressions, it simultaneously marginalizes much of the music that is “just using computers” and fails to provide insights into this divided landscape.

Lyon argues that defining computer music simply as music created with computers is too permissive, while defining it as music that could not exist without computers is too strict. He highlights the difficulty of considering instruments that use digital simulations, such as virtual analog synthesizers, within these definitions. Furthermore, he suggests that the term "computer music" is style-agnostic definition almost like "piano music," implying that it ignores the style and form inside music produced by the instruments.

However, one of the defining characteristics of computers as a medium lies in their ability to treat musical styles themselves as subjects of meta-manipulation through simulation and modeling. When creating instruments with computers, or when using such instruments, sound production involves programming—manipulating symbols embedded in a particular musical culture. This recursive embedding of the language and perception constituting that musical culture into the resulting music is a process that goes beyond what is possible with acoustic instruments or analog electronic instruments. Magnusson refers to this characteristic of digital instruments as "Epistemic Tools" and points out that they tend to work in the direction of reinforcing and solidifying musical culture:

> The act of formalising is therefore always an act of fossilisation. As opposed to the acoustic instrument maker, the designer of the composed digital instrument frames affordances through symbolic design, thereby creating a snapshot of musical theory, freezing musical culture in time. [11, p173]

Today, many people use computers for music production not because they consciously leverage the uniqueness of the meta-medium, but simply because there are no quicker or more convenient alternatives available. Even so, within a musical culture where computers are used as a default or reluctant choice, musicians are inevitably influenced by the underlying infrastructures like software, protocols, and formats. As long as the history of programming languages for music remains intertwined with the history of computer music as it relates to specific genres or communities, it becomes difficult to analyze music created with computers as a passive means.

In this paper, the history of programming languages for music is reexamined with an approach that, opposite from Lyon, takes an extremely style-agnostic perspective. Rather than focusing on what has been created with these tools, the emphasis is placed on how these tools themselves have been constructed. The paper centers on the following two topics:

1. A critique of the universality of sound representation using pulse-code modulation (PCM), the foundational concept underlying most of today's sound programming, by referencing early attempts of sound generation using electronic computers.
2. An examination of the MUSIC-N family, the origin of PCM-based sound synthesis, to highlight that its design varies significantly across systems from the perspective of modern programming language design and that it has evolved over time into a black box, eliminating the need for users to understand its internal workings.

Ultimately, the paper concludes that programming languages for music developed since the 2000s are not solely aimed at creating new music but also serve as alternatives to the often-invisible technological infrastructures surrounding music, such as formats and protocols. By doing so, the paper proposes new perspectives for the historical study of music created with computers. ## PCM and Early Computer Music

Among the earliest examples of computer music research, the MUSIC I system (1957) from Bell Labs and its derivatives, known as MUSIC-N, are frequently highlighted. However, attempts to create music with computers in the UK and Australia prior to MUSIC I have also been documented [12]. Organizing what was achieved by MUSIC-N and earlier efforts can help clarify definitions of computer music.

The earliest experiments with sound generation on computers in the 1950s involved controlling the intervals between one-bit pulses (on or off) to control pitch. This was partly because the operational clock frequencies of early computers fell within the audible range, making the sonification of electrical signals a practical and cost-effective debugging method compared to visualizing them on displays or oscilloscopes. Some computers at this time like Australia's CSIR Mark I (CSIRAC) often had "hoot" primitive instructions that emit a single pulse to a speaker.

In 1949, the background to music played on the BINAC in UK involved engineer Louis Wilson, who noticed that an AM radio placed nearby could pick up weak electromagnetic waves generated during the switching of vacuum tubes, producing regular sounds. He leveraged this phenomenon by connecting a speaker and a power amplifier to the computer's output, using the setup to assist in debugging processes. Frances Elizabeth Holberton took this a step further by programming the computer to generate pulses at arbitrary intervals, creating melodies [13]. The sound generation on BINAC and CSIR Mark I represents early instances of using computers to play melodies from existing music.

However, not all sound generation at this timewas merely the reproduction of existing music. Doornbusch highlights experiments on the British Pilot ACE (Prototype for Automatic Computing Engine: ACE), which utilized acoustic delay line memory to produce unique sounds [12, p303-304]. Acoustic delay line memory, used as main memory in early computers like BINAC and CSIR Mark I, employed the feedback of pulses traveling through mercury via a speaker and microphone setup to retain data. Donald Davis, an engineer on the ACE project, described the sounds it produced as follows [14, p19-20]:

> The Ace Pilot Model and its successor, the Ace proper, were both capable of composing their own music and playing it on a little speaker built into the control desk. I say composing because no human had any intentional part in choosing the notes. The music was very interesting, though atonal, and began by playing rising arpeggios: these gradually became more complex and faster, like a developing fugue. They dissolved into colored noise as

> the complexity went beyond human understanding.
>
> Loops were always multiples of 32 microseconds long, so notes had frequencies which were submultiples of 31.25 KHz. The music was based on a very strange scale, which was nothing like equal tempered or harmonic, but was quite pleasant.

This music arose unintentionally during program optimization and was made possible by "misusing" switches installed for debugging acoustic delay line memory (p20). Media scholar Miyazaki described the practice of listening to sounds generated by algorithms and their bit patterns, integrated into programming and debugging, as "Algo*rhythmic* Listening" [15].

Doornbusch warns against ignoring early computer music practices in Australia and the UK simply because they did not directly influence subsequent research [12, p305]. Indeed, the tendency to treat pre-MUSIC attempts as hobbyist efforts by engineers and post-MUSIC endeavors as "serious" research remains common even today [16].

The sounds produced by the Pilot ACE challenge the post-acousmatic historical narrative, which suggests that computer music transitioned from being confined to specialized laboratories to becoming accessible to individuals, including amateurs.

This is because the sounds generated by the Pilot ACE were not created by musical experts, nor were they solely intended for debugging purposes. Instead, they were programmed with the goal of producing interesting sounds. Moreover, the sounds were tied to the hardware of the acoustic delay line memory—a feature that was likely difficult to replicate, even in modern audio programming environments.

Similarly, in the 1960s at MIT, Peter Samson took advantage of the debugging speaker on the TX-0, a machine that had become outdated and freely available for students to use. He conducted experiments where he played melodies, such as Bach fugues, using square waves [17]. Samson's experiments with the TX-0 later evolved into the creation of a program that allowed melodies to be described using text strings within MIT.

Building on this, Samson developed a program called the Harmony Compiler on the DEC PDP-1, which was derived from the TX-0. This program gained significant popularity among MIT students. Around 1972, Samson began surveying various digital synthesizers that were being developed at the time and went on to create a system specialized for computer music. The resulting Samson Box was used at Stanford University's CCRMA (Center for Computer Research in Music and Acoustics) for over a decade until the early 1990s and became a tool for many composers to create their works [18]. Considering Samson's example, it is not appropriate to separate the early experiments in sound generation by computers from the history of computer music solely because their initial purpose was debugging. ### Acousmatic Listening, the premise of the Universality of PCM

One of the reasons why MUSIC led to subsequent advancements in research was not simply because it was developed early, but because it was the first to implement sound representation on a computer based on **pulse-code modulation (PCM)**, which theoretically enables the representation of "almost any sound."

PCM, the foundational method of sound representation on today's computers, involves dividing audio waveforms into discrete intervals (sampling) and representing the sound pressure at each interval as discrete numerical values (quantization).

The issue with the universalism of PCM in the history of computer music is inherent in the concept of Acousmatic, which serves as a premise for Post-Acousmatic. Acousmatic, introduced by Piegnot as a listening style for tape music such as musique concrète and later theorized by Schaeffer, refers to a mode of listening where the listener refrains from imagining a specific sound source. This concept has been widely applied in theories of listening to recorded sound, including Chion's analysis of sound design in film.

However, as sound studies scholar Jonathan Sterne has pointed out, discourses surrounding acousmatic listening often work to delineate pre-recording auditory experiences as "natural" by contrast [1]. This implies that prior to the advent of recording technologies, listening was unmediated and holistic—a narrative that obscures the constructed nature of these assumptions.

> For instance, the claim that sound reproduction has "alienated" the voice from the human body implies that the voice and the body existed in some prior holistic, unalienated, and self present relation.
>
> They assume that, at some time prior to the invention of sound reproduction technologies, the body was whole, undamaged, and phenomenologically coherent. [20, p20-21]

The claim that PCM-based sound synthesis can produce "almost any sound" is underpinned by an ideology associated with recording technologies. This ideology assumes that recorded sound contains an "original" source and that listeners can distinguish distortions or noise from it. Sampling theory builds on this premise by statistically modeling human auditory characteristics: it assumes that humans cannot discern volume differences below certain thresholds or perceive vibrations outside specific frequency ranges. By limiting representation to this range, sampling theory ensures that all audible sounds can be effectively encoded.

By the way, the actual implementation of PCM in MUSIC I only allowed for monophonic triangle waves with controllable volume, pitch, and timing (MUSIC II later expanded this to four oscillators) [21]. Would anyone today describe such a system as capable of producing "infinite variations" in sound synthesis?

Even when considering more contemporary applications, processes like ring modulation (RM), amplitude modulation (AM), or distortion often generate aliasing artifacts

[1] Sterne later critiques the phenomenological basis of acousmatic listening, which presupposes an idealized, intact body as the listening subject. He proposes a methodology of political phenomenology centered on impairment, challenging these normative assumptions [19]. Discussions of universality in computer music should also address ableism, as seen in the relationship between recording technologies and auditory disabilities.

unless proper oversampling is applied. These artifacts occur because PCM, while universally suitable for reproducing recorded sound, is not inherently versatile as a medium for generating new sounds. As Puckette has argued, alternative representations, such as collections of linear segments or physical modeling synthesis, present other possibilities [22]. Therefore, PCM is not a completely universal tool for creating sound.

2. WHAT DOES THE UNIT GENERATOR HIDE?

Starting with version III, MUSIC adopted the form of an acoustic compiler (or block diagram compiler) that takes two types of input: a score language, which represents a list of time-varying parameters, and an orchestra language, which describes the connections between **Unit Generators** such as oscillators and filters. In this paper, the term "Unit Generator" means a signal processing module used by the user, where the internal implementation is either not open or implemented in a language different from the one used by the user.

Beyond performing sound synthesis based on PCM, one of the defining features of the MUSIC family in the context of computer music research was the establishment of a division of labor between professional musicians and computer engineers through the development of domain-specific languages. Mathews explained that he developed a compiler for MUSIC III in response to requests for additional features such as envelopes and vibrato, while also ensuring that the program would not be fixed in a static form [23, 13:10-17:50]. He repeatedly stated that his role was that of a scientist rather than a musician:

> The only answer I could see was not to make the instruments myself—not to impose my taste and ideas about instruments on the musicians—but rather to make a set of fairly universal building blocks and give the musician both the task and the freedom to put these together into his or her instruments. [21, p16]
> (...) When we first made these music programs the original users were not composers; they were the psychologist Guttman, John Pierce, and myself, who are fundamentally scientists. We wanted to have musicians try the system to see if they could learn the language and express themselves with it. So we looked for adventurous musicians and composers who were willing to experiment. (p17)

This clear delineation of roles between musicians and scientists became one of the defining characteristics of post-MUSIC computer music research. Paradoxically, the act of creating sounds never heard before using computers paved the way for research by allowing musicians to focus on their craft without needing to grapple with the complexities of programming.

2.1 Example: Hiding First-Order Variables in Signal Processing

Although the MUSIC N series shares a common workflow of using a Score language and an Orchestra language, the actual implementation of each programming language varies significantly, even within the series.

One notable but often overlooked example is MUSIGOL, a derivative of MUSIC IV [24]. In MUSIGOL, not only was the system itself implemented differently, but even the user-written Score and Orchestra programs were written entirely as ALGOL 60 source code. Similar to modern frameworks like Processing or Arduino, MUSIGOL represents one of the earliest examples of a domain-specific language implemented as an internal DSL within a library [2]. (Therefore, according to the definition of Unit Generator provided in this paper, MUSIGOL does not qualify as a language that uses Unit Generators.)

The level of abstraction deemed intuitive for musicians varied across different iterations of the MUSIC N series. This can be illustrated by examining the description of a second-order band-pass filter. The filter mixes the current input signal S_n, the output signal from t time steps prior O_{n-t}, and an arbitrary amplitude parameter I_1, as shown in the following equation:

$$O_n = I_1 \cdot S_n + I_2 \cdot O_{n-1} - I_3 \cdot O_{n-2}$$

In MUSIC V, this band-pass filter can be used as in 1 [26, p78].

```
FLT I1 O I2 I3 Pi Pj;
```

Listing 1. Example of the use of RESON UGen in MUSIC V.

Here, `I1` represents the input bus, and `O` is the output bus. The parameters `I2` and `I3` correspond to the normalized values of the coefficients I_2 and I_3, divided by I_1 (as a result, the overall gain of the filter can be greater or less than 1). The parameters `Pi` and `Pj` are normally used to receive parameters from the Score, specifically among the available `P0` to `P30`. In this case, however, these parameters are repurposed as general-purpose memory to temporarily store feedback signals. Similarly, other Unit Generators, such as oscillators, reuse note parameters to handle operations like phase accumulation.

As a result, users needed to manually calculate feedback gains based on the desired frequency characteristics [3], and they also had to account for using at least two sample memory spaces.

On the other hand, in MUSIC 11, developed by Barry Vercoe, and its later iteration, CSound, the band-pass filter is defined as a Unit Generator (UGen) named `reson`. This UGen accepts four parameters: the input signal, center cutoff frequency, bandwidth, and Q factor. Unlike previous implementations, users no longer need to be aware of the two-sample feedback memory space for the output [27, p248]. However, in MUSIC 11 and CSound, it is still possible to implement this band-pass filter from scratch as a User Defined Opcode (UDO) as in 2. Vercoe emphasized that while signal processing primitives should allow for low-level operations, such as single-sample feedback, and eliminate black boxes, it is equally important to provide high-level modules that avoid unnecessary complexity

[2] While MUS10, used at Stanford University, was not an internal DSL, it was created by modifying an existing ALGOL parser [25, p248].

[3] It is said that a preprocessing feature called `CONVT` could be used to transform frequency characteristics into coefficients [26, p77].

("avoid the clutter") when users do not need to understand the internal details [27, p247].

```
     instr  1
la1  init   0
la2  init   0
i3   = exp(-6.28 * p6 / 10000)
i2   = 4*i3*cos(6.283185 * p5/10000) / (1+
   i3)
i1   = (1-i3) * sqrt(1-1 - i2*i2/(4*i3))
a1   rand   p4
la3  =      la2
la2  =      la1
la1  =      i1*a1 + i2 * la2 - i3 * la3
     out    la1
     endin

     instr 2
a1   rand   p4
a1   reson  a1,p5,p6,1
     endin
```

Listing 2. Example of scratch implementation and built-in operation of RESON UGen respectively, in MUSIC11. Retrieved from the original paper. (Comments are omitted for the space restriction.)

On the other hand, in programming environments that inherit the Unit Generator paradigm, such as Pure Data [28], Max (whose signal processing functionalities were ported from Pure Data as MSP), SuperCollider [29], and ChucK [**?**], primitive UGens are implemented in general-purpose languages like C or C++. If users wish to define low-level UGens (External Objects), they need to set up a development environment for C or C++.

As an extension, ChucK later introduced ChuGen, which is equivalent to CSound's UDO, allowing users to define low-level UGens within the ChucK language itself [30]. However, both CSound and ChucK face performance limitations with UDOs during runtime compared to natively implemented UGens. Consequently, not all existing UGens are replaced by UDOs, which remain supplemental features rather than primary tools.

When UGens are implemented in low-level languages like C, even if the implementation is open-source, the division of knowledge effectively forces users (composers) to treat UGens as black boxes. This reliance on UGens as black boxes reflects and deepens the division of labor between musicians and scientists that Mathews helped establish—a structure that can be seen as both a cause and a result of this paradigm.

For example, Puckette, the developer of Max and Pure Data, noted that the division of labor at IRCAM between researchers, Musical Assistants/realizers, and composers has parallels in the current Max ecosystem, where the roles are divided into software developers, External Objects developers, and Max users [31]. As described in the ethnography of 1980s IRCAM by anthropologist Georgina Born, the division of labor between fundamental research scientists and composers at IRCAM was extremely clear. This structure was also tied to the exclusion of popular music and its associated technologies in IRCAM's research focus [32].

However, such divisions are not necessarily the result of differences in values along the axes analyzed by Born, such as modernist/postmodernist/populist or low-tech/high-tech distinctions [4]. This is because the black-boxing of technology through the division of knowledge occurs in popular music as well. Paul Théberge pointed out that the "democratization" of synthesizers in the 1980s was achieved through the concealment of technology, which transformed musicians as creators into consumers.

> Lacking adequate knowledge of the technical system, musicians increasingly found themselves drawn to prefabricated programs as a source of new sound material. As I have argued, however, this assertion is not simply a state ment of fact; it also suggests a reconceptualization on the part of the industry of the musician as a particular type of consumer. [34, p89]

This argument can be extended beyond electronic music to encompass computer-based music in general. For example, media researcher Lori Emerson noted that while the proliferation of personal computers began with the vision of "metamedia"—tools that users could modify themselves, as exemplified by Xerox PARC's Dynabook—the vision was ultimately realized in an incomplete form through devices like the Macintosh and iPad, which distanced users from programming by black-boxing functionality [35]. In fact, Alan Kay, the architect behind the Dynabook concept, remarked that while the iPad's appearance may resemble the ideal he originally envisioned, its lack of extensibility through programming renders it merely a device for media consumption [36].

Although programming environments as tools for music production are not widely used, the Unit Generator concept, alongside MIDI, serves as a foundational paradigm for today's consumer music production software and infrastructure, including Web Audio. It is known that the concept of Unit Generators emerged either simultaneously with or even slightly before modular synthesizers [**?**, p20]. However, UGen-based languages have actively incorporated the user interface metaphors of modular synthesizers, as Vercoe said that the distinction between "ar" (audio-rate) and "kr" (control-rate) processing introduced in MUSIC 11 is said to have been inspired by Buchla's differentiation between control and audio signals in its plug type [**?**, 1:01:38–1:04:04].

However, adopting visual metaphors comes with the limitation that it constrains the complexity of representation to what is visually conceivable. In languages with visual patching interfaces like Max and Pure Data, meta-operations on UGens are often restricted to simple tasks, such as parallel duplication. Consequently, even users of Max or Pure Data may not necessarily be engaging in expressions that are only possible with computers. Instead, many might simply be using these tools as the most convenient software equivalents of modular synthesizers.

[4] David Wessel revealed that the individual referred to as RIG in Born's ethnography was himself and commented that Born oversimplified her portrayal of Pierre Boulez, then director of IRCAM, as a modernist. [33]

3. CONTEXT OF PROGRAMMING LANGUAGES FOR MUSIC AFTER 2000

Based on the discussions thus far, music programming languages developed after the 2000s can be categorized into two distinct directions: those that narrow the scope of the language's role by attempting alternative abstractions at a higher level, distinct from the Unit Generator paradigm, and those that expand the general-purpose capabilities of the language, reducing black-boxing.

Languages that pursued alternative abstractions at higher levels have evolved alongside the culture of live coding, where performances are conducted by rewriting code in real time. The activities of the live coding community, including groups such as TOPLAP since the 2000s, were not only about turning coding itself into a performance but also served as a resistance against laptop performances that relied on black-boxed music software. This is evident in the community's manifesto, which states, "Obscurantism is dangerous" [37].

Languages implemented as clients for SuperCollider, such as **IXI** (on Ruby) [38], **Sonic Pi**(on Ruby), **Overtone** (on Clojure) [39], **TidalCycles** (on Haskell) [40], and **FoxDot** (on Python) [41], leverage the expressive power of more general-purpose programming languages. While embracing the UGen paradigm, they enable high-level abstractions for previously difficult-to-express elements like note values and rhythm. For example, the abstraction of patterns in TidalCycles is not limited to music but can also be applied to visual patterns and other outputs, meaning it is not inherently tied to PCM-based waveform output as the final result.

On the other hand, due to their high-level design, these languages often rely on ad hoc implementations for tasks like sound manipulation and low-level signal processing, such as effects.

McCartney, the developer of SuperCollider, once stated that if general-purpose programming languages were sufficiently expressive, there would be no need to create specialized languages [1]. This prediction appears reasonable when considering examples like MUSIGOL. However, in practice, scripting languages that excel in dynamic program modification face challenges in modern preemptive OS environments. For instance, dynamic memory management techniques such as garbage collection can hinder the ability to guarantee deterministic execution timing required for real-time processing [**?**].

Historically, programming in languages like FORTRAN or C served as a universal method for implementing audio processing on computers, independent of architecture. However, with the proliferation of general-purpose programming languages, programming in C or C++ has become relatively more difficult, akin to programming in assembly language in earlier times. Furthermore, considering the challenges of portability across not only different CPUs but also diverse host environments such as operating systems and the Web, these languages are no longer as portable as they once were. Consequently, systems targeting signal processing implemented as internal DSLs have become exceedingly rare, with only a few examples like LuaAV [42].

Instead, an approach has emerged to create general-purpose languages specifically designed for use in music from the ground up. One prominent example is **Extempore**, a live programming environment developed by Sorensen [**?**]. Extempore consists of Scheme, a LISP-based language, and xtlang, a meta-implementation on top of Scheme. While xtlang requires users to write hardware-oriented type signatures similar to those in C, it leverages the LLVM compiler infrastructure [43] to just-in-time (JIT) compile signal processing code, including sound manipulation, into machine code for high-speed execution.

The expressive power of general-purpose languages and compiler infrastructures like LLVM have given rise to an approach focused on designing languages with formalized abstractions that reduce black-boxing. **Faust** [44], for example, is a language that retains a graph-based structure akin to UGens but is built on a formal system called Block Diagram Algebra. This system integrates primitives for reading and writing internal states, which are essential for operations like delays and filters. Thanks to its formalization, Faust can be transpiled into general-purpose languages such as C, C++, or Rust and can also be used as an External Object in environments like Max or Pure Data.

Languages like **Kronos** [**?**] and **mimium** [**?**], which are based on the more general computational model of lambda calculus, focus on PCM-based signal processing while exploring interactive meta-operations on programs [**?**] and balancing self-contained semantics with interoperability with other general-purpose languages [**?**].

Domain-specific languages (DSLs) are constructed within a double bind: they aim to specialize in a particular purpose while still providing a certain degree of expressive freedom through programming. In this context, efforts like Extempore, Kronos, and mimium are not merely programming languages for music but are also situated within the broader research context of Functional Reactive Programming (FRP), which focuses on representing time-varying values in computation. Most computer hardware lacks an inherent concept of real time and instead operates based on discrete computational steps. Similarly, low-level general-purpose programming languages do not natively include primitives for real-time concepts. Consequently, the exploration of computational models tied to time—a domain inseparable from music—remains vital and has the potential to contribute to the theoretical foundations of general-purpose programming languages.

However, strongly formalized languages come with their own trade-offs. While they allow UGens to be defined without black-boxing, understanding the design and implementation of these languages often requires advanced knowledge. This can create a significant divide between language developers and users, in contrast to the more segmented roles seen in the Multi-Language paradigm—such as SuperCollider developers, external UGen developers, client language developers (e.g., TidalCycles), SuperCollider users, and client language users.

Although there is no clear solution to this trade-off, one intriguing idea is the development of self-hosting languages for music—that is, languages where their own compilers are written in the language itself. At first glance, this may seem impractical. However, by enabling users to learn and modify the language's mechanisms spontaneously, this approach could create an environment that fosters deeper engagement and understanding among users.

4. CONCLUSION

This paper has reexamined the history of computer music and music programming languages with a focus on the universalism of PCM and the black-boxing tendencies of the Unit Generator paradigm. Historically, it was expected that the clear division of roles between engineers and composers would enable the creation of new forms of expression using computers. Indeed, from the perspective of Post-Acousmatic discourse, some, like Holbrook and Rudi, still consider this division to be a positive development:

> Most newer tools abstract the signal processing routines and variables, making them easier to use while removing the need for understanding the underlying processes in order to create meaningful results. Composers no longer necessarily need mathematical and programming skills to use the technologies. These abstractions are important, as they hide many of the technical details and make the software and processes available to more people, and form the basis for what can arguably be seen as a new folk music. [10, p2]

However, this division of labor also creates a shared vocabulary—exemplified by the Unit Generator itself, pioneered by Mathews—and works to perpetuate it. By portraying new technologies as something externally introduced, and by focusing on the agency of those who create music with computers, the individuals responsible for building the programming environments, software, protocols, and formats are rendered invisible [45]. This leads to an oversight of the indirect power relationships produced by these infrastructures.

For this reason, future research on programming languages for music must address how the tools, including the languages themselves, contribute aesthetic value within musical culture (and what forms of musical practice they enable), as well as the social (im)balances of power they produce.

It has been noted in programming language research that evaluation criteria such as efficiency, expressiveness, and generality are often ambiguous [46]. This issue is even more acute in fields like music, where no clear evaluation criteria exist. Thus, as McPherson et al. have proposed with the concept of Idiomaticity [3], we need to develop and share a vocabulary for understanding the value judgments we make about programming languages in general.

In a broader sense, the creation of programming languages for music has also expanded to the individual level. Examples include **Gwion** by Astor, which builds on ChucK and enhances its abstraction capabilities with features like lambda functions [**?**]; **Vult**, a DSP transpiler language created by Ruiz for his modular synthesizer hardware [**?**]; and a UGen-based live coding environment designed for web execution, **Glicol** [47]. However, these efforts have not yet been adequately integrated into academic discourse.

Conversely, practical knowledge of university-researched languages from the past, as well as real-time hardware-oriented systems from the 1980s, is gradually being lost. While research efforts such as *Inside Computer Music*, which analyzes historical works of computer music, have begun [**?**], an archaeological practice focused on the construction of computer music systems will also be necessary in the future. This includes not only collecting primary resources, such as oral archives from those involved, but also reconstructing the knowledge and practices behind these systems.

Acknowledgments

At the end of the Conclusions, acknowledgments to people, projects, funding agencies, etc. can be included after the second-level heading "Acknowledgments" (with no numbering).

5. REFERENCES

[1] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, Dec. 2002.

[2] H. Nishino and R. Nakatsu, "Computer Music Languages and Systems: The Synergy Between Technology and Creativity," in *Handbook of Digital Games and Entertainment Technologies*, 2016.

[3] A. McPherson and K. Tahlroğlu, "Idiomatic Patterns and Aesthetic Influence in Computer Music Languages," *Organised Sound*, vol. 25, no. 1, pp. 53–63, 2020.

[4] R. B. Dannenberg, "Languages for Computer Music," *Frontiers in Digital Humanities*, vol. 5, Nov. 2018.

[5] V. Lazzarini, "The Development of Computer Music Programming Systems," *Journal of New Music Research*, vol. 42, no. 1, pp. 97–110, 2013.

[6] M. V. Mathews, "An Acoustic Compiler for Music and Psychological Stimuli," *The Bell System Technical Journal*, vol. 40, no. 3, pp. 677–694, May 1961.

[7] B. Ostertag, "Why Computer Music Sucks," https://web.archive.org/web/20160312125123/http://bobostertag.co articles-computer-music-sucks.htm, 1998.

[8] E. Lyon, "Do We Still Need Computer Music?" in *EMS*, 2006.

[9] M. Adkins, R. Scott, and P. A. Tremblay, "Post-Acousmatic Practice: Re-evaluating Schaeffer's Heritage," *Organised Sound*, vol. 21, no. 2, pp. 106–116, Aug. 2016.

[10] U. Holbrook and J. Rudi, "Computer Music and Post-Acousmatic Practices: International Computer Music Conference 2022," in *Proceedings of the International Computer Music Conference, ICMC 2022*, ser. International Computer Music Conference, ICMC Proceedings, G. Torre, Ed. San Francisco: International Computer Music Association, Jul. 2022, pp. 140–144.

[11] T. Magnusson, "Of Epistemic Tools: Musical Instruments as Cognitive Extensions," *Organised Sound*, vol. 14, no. 2, pp. 168–176, Aug. 2009.

[12] P. Doornbusch, "Early Computer Music Experiments in Australia and England," *Organised Sound*, vol. 22, no. 2, pp. 297–307, Aug. 2017.

[13] R. D. Woltman, F. B. Woltman, L. D. Wilson, A. B. Tonik, J. K. Swearingen, C. M. Shuler, J. E. Sberro, J. E. Sammet, H. W. Matter, D. W. Marquardt, F. K. Koons, M. W. Huff, F. E. Holberton, C. Hammer, D. B. Dixon, E. L. Delves, G. Danehower, M. P. Chinitz, L. S. Carter, J. Bartik, L. W. Armstrong, D. P. Armstrong, and A. E. Adams, "UNIVAC Conference." Charles Babbage Institute, Tech. Rep., 1990.

[14] D. Davis, "Very Early Computer Music," *Resurrection The Bulletin of the Computer Conservation Society*, vol. 10, pp. 19–20, 1994.

[15] S. Miyazaki, "Algorhythmic Listening 1949-1962 Auditory Practices of Early Mainframe Computing," in *AISB/IACAP World Congress 2012: Symposium on the History and Philosophy of Programming, Part of Alan Turing Year 2012*, 2012, p. 5.

[16] H. Tanaka, *All About Chiptune: New Music Born from Games*. Seibundo Shinkosha, 2017.

[17] S. Levy, *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*, 1st ed. O'Reilly Media, May 2010.

[18] D. G. Loy, "Life and Times of the Samson Box," *Computer Music Journal*, vol. 37, no. 3, pp. 26–48, 2013.

[19] J. Sterne, *Diminished Faculties: A Political Phenomenology of Impairment*. Durham: Duke Univ Press, Jan. 2022.

[20] ——, *The Audible Past: Cultural Origins of Sound Reproduction*. Durham: Duke University Press, 2003.

[21] M. Mathews and C. Roads, "Interview with Max Mathews," *Computer Music Journal*, vol. 4, no. 4, pp. 15–22, 1980.

[22] M. Puckette, "The Sampling Theorem and Its Discontents," *International Computer Music Conference*, pp. 1–14, 2015.

[23] M. V. Mathews, "Max Mathews Full Interview | NAMM.Org," https://www.namm.org/video/orh/max-mathews-full-interview, Mar. 2007.

[24] D. M. Innis, "Sound Synthesis by Computer: Musigol, a Program Written Entirely in Extended Algol," *Perspectives of New Music*, vol. 7, no. 1, pp. 66–79, 1968.

[25] G. Loy and C. Abbott, "Programming Languages for Computer Music Synthesis, Performance, and Composition," *ACM Comput. Surv.*, vol. 17, no. 2, pp. 235–265, Jun. 1985.

[26] M. V. Mathews and J. E. Miller, *The Technology of Computer Music*. M.I.T. Press, 1969.

[27] B. L. Vercoe, "Computer Systems and Languages for Audio Research," *The New World of Digital Audio (Audio Engineering Society Special Edition)*, pp. 245–250, 1983.

[28] M. Puckette, "Pure Data," in *International Computer Music Conference Proceedings*. Michigan Publishing, University of Michigan Library, 1997.

[29] J. McCartney, "SuperCollider, a New Real Time Synthesis Language," in *International Computer Music Conference Proceedings*. Michigan Publishing, 1996.

[30] S. Salazar and G. Wang, "CHUGENS, CHUBGRAPHS, CHUGINS: 3 TIERS FOR EXTENDING CHUCK," in *International Computer Music Conference Proceedings*, 2012, pp. 60–63.

[31] I. Reese, "47 • Miller Puckette • Max/MSP & Pure Data," May 2020.

[32] G. Born, *Rationalizing Culture*. University of California Press, 1995, no. 1.

[33] G. Taylor, "Article: An Interview With David Wessel | Cycling '74," https://cycling74.com/articles/an-interview-with-david-wessel, 1999.

[34] P. Théberge, *Any Sound You Can Imagine: Making Music/Consuming Technology*, ser. Music/Culture. Hanover, NH: Wesleyan University Press : University Press of New England, 1997.

[35] L. Emerson, *Reading Writing Interfaces: From the Digital to the Bookbound*. Univ of Minnesota Press, Nov. 2014.

[36] A. C. Kay, "American Computer Pioneer Alan Kay's Concept, the Dynabook, Was Published in 1972. How Come Steve Jobs and Apple iPad Get the Credit for Tablet Invention?" https://www.quora.com/American-computer-pioneer-Alan-Kay-s-concept-the-Dynabook-was-published-in-1972-How-come-Steve-Jobs-and-Apple-iPad-get-the-credit-for-tablet-invention, Apr. 2019.

[37] TOPLAP, "ManifestoDraft - Toplap," https://toplap.org/wiki/ManifestoDraft, 2004.

[38] T. Magnusson, "The IXI Lang: A SuperCollider Parasite for Live Coding," *International Computer Music Conference Proceedings*, vol. 2011, 2011.

[39] S. Aaron and A. F. Blackwell, "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pp. 35–46, 2013.

[40] A. McLean, "Making Programming Languages to Dance to: Live Coding with Tidal," in *FARM 2014 - Proceedings of the 2014 ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. New York, New York, USA: Association for Computing Machinery, 2014, pp. 63–70.

[41] R. Kirkbride, "FoxDot: Live Coding with Python and Supercollider," in *Proceedings of the International Conference on Live Interfaces*, 2016, pp. 194–198.

[42] G. Wakefield, W. Smith, and C. Roberts, "LuaAV: Extensibility and Heterogeneity for Audiovisual Computing," in *Proceeding of Linux Audio Conference*, 2010.

[43] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2004, p. 75.

[44] Y. Orlarey, D. Fober, S. Letz, and S. Letz, “FAUST : An Efficient Functional Approach to DSP Programming,” in *New Computational Paradigms for Computer Music*. DELATOUR FRANCE, 2009.

[45] J. Sterne, “There Is No Music Industry,” *Media Industries Journal*, vol. 1, no. 1, pp. 50–55, Jan. 2014.

[46] S. Markstrum, “Staking Claims: A History of Programming Language Design Claims and Evidence: A Positional Work in Progress,” *Evaluation and Usability of Programming Languages and Tools, PLATEAU'10*, 2010.

[47] Q. Lan, “Glicol,” https://glicol.org/, 2020.