

NAVIGATING THE LANDSCAPE OF COMPUTER AIDED ALGORITHMIC COMPOSITION SYSTEMS: A DEFINITION, SEVEN DESCRIPTORS, AND A LEXICON OF SYSTEMS AND RESEARCH

Christopher Ariza

New York University Graduate
School of Arts and Sciences
New York, New York
ariza@flexatone.net

ABSTRACT

Towards developing methods of software comparison and analysis, this article proposes a definition of a computer aided algorithmic composition (CAAC) system and offers seven system descriptors: scale, process-time, idiom-affinity, extensibility, event production, sound source, and user environment. The public internet resource *algorithmic.net* is introduced, providing a lexicon of systems and research in computer aided algorithmic composition.

1. DEFINITION OF A COMPUTER-AIDED ALGORITHMIC COMPOSITION SYSTEM

Labels such as algorithmic composition, automatic composition, composition pre-processing, computer-aided composition (CAC), computer composing, computer music, procedural composition, and score synthesis have all been used to describe overlapping, or sometimes identical, projects in this field. No attempt will be made to distinguish these terms, though some have tried (Spiegel 1989; Cope 1991, p. 220; Burns 1994, p. 195; Miranda 2000, pp. 9-10; Taube 2004; Gerhard and Hepting 2004, p. 505). In order to provide greater specificity, a hybrid label is introduced: CAAC, or computer aided algorithmic composition. (This term is used in passing by Martin Supper (2001, p. 48).) This label is derived from the combination of two labels, each too vague for continued use. The label “computer aided composition” lacks the specificity of using generative algorithms. Music produced with notation or sequencing software could easily be considered computer aided composition. The label “algorithmic composition” is likewise too broad, particularly in that it does not specify the use of a computer. Although Mary Simoni has suggested that “because of the increased role of the computer in the compositional process, algorithmic composition has come to mean the use of computers...” (2003), there remain many historical and contemporary compositional techniques that, while not employing the computer, are properly described as algorithmic. David Cope supports this view, stating that “... the term ‘computer’ is not requisite to a definition of algorithmic composition...” (1993, p. 24).

Since 1955 a wide variety of CAAC systems have been created. Towards the aim of providing tools for software

comparison and analysis, this article proposes seven system descriptors. Despite Lejaren Hiller’s well-known claim that “computer-assisted composition is difficult to define, difficult to limit, and difficult to systematize” (Hiller 1981, p. 75), a definition is proposed.

A CAAC system is software that facilitates the generation of new music by means other than the manipulation of a direct music representation. Here, “new music” does not designate style or genre; rather, the output of a CAAC system must be, in some manner, a unique musical variant. An output, compared to the user’s representation or related outputs, must not be a “copy,” accepting that the distinction between a copy and a unique variant may be vague and contextually determined. This output may be in the form of any sound or sound parameter data, from a sequence of samples to the notation of a complete composition. A “direct music representation” refers to a linear, literal, or symbolic representation of complete musical events, such as an event list (a score in Western notation or a MIDI file) or an ordered list of amplitude values (a digital audio file or stream). Though all representations of aural entities are necessarily indirect to some degree, the distinction made here is not between these representations and aural entities. Rather, a distinction is made between the representation of musical entities provided to the user and the system output. If the representation provided to the user is the same as the output, the representation may reasonably be considered direct.

A CAAC system permits the user to manipulate indirect musical representations: this may take the form of incomplete musical materials (a list of pitches or rhythms), an equation, non-music data, an image, or meta-musical descriptions. Such representations are indirect in that they are not in the form of complete, ordered musical structures. In the process of algorithmic generation these indirect representations are mapped or transformed into a direct music representation for output. When working with CAAC software, the composer arranges and edits these indirect representations. The software interprets these indirect music representations to produce musical structures.

This definition does not provide an empirical measure by which a software system, removed from use, can be isolated as a CAAC system. Rather, a contextual

delineation of scope is provided, based in part on use case. Consideration must be given to software design, functionality, and classes of user interaction.

This definition is admittedly broad, and says only what a CAAC system is not. This definition includes historic systems such as the Experiments of Hiller and Isaacson (1959), Iannis Xenakis's SMP (1965), and Gottfried Michael Koenig's PR1 (1970a) and PR2 (1970b). In these cases the user provides initial musical and non-musical data (parameter settings, value ranges, stockpile collections), and these indirect representations are mapped into score tables. This definition likewise encompasses Xenakis's GENDYN (1992) and Koenig's SSP (Berg et al 1980). This definition includes any system that converts images (an indirect representation) to sound, such as Max Mathews and L. Rosler's Graphic 1 system (1968) or Xenakis's UPIC (1992; Marino et al. 1993). It does not matter how the images are made; they might be from a cellular automaton, a digital photograph, or hand-drawn. What matters is that the primary user-interface is an indirect representation. Some systems may offer the user both direct and indirect music representations. If one representation is primary, that representation may define the system; if both representations are equally presented to the user, a clear distinction may not be discernible.

This definition excludes, in most use cases, notation software. Notation software is primarily used for manipulating and editing a direct music representation, namely Western notation. New music is not created by notation software: the output, the score, is the user-defined representation. Recently, systems such as the popular notation applications Sibelius (Sibelius Software Limited) and Finale (MakeMusic! Inc.) have added user-level interfaces for music data processing in the form of specialized scripting languages or plug-ins. These tools allow the user to manipulate and generate music data as notation. In this case, the script and its parameters are an indirect music representation and can be said to have attributes of a CAAC system. This is not, however, the primary user-level interface.

This definition excludes, in most use cases, digital audio workstations, sequencers, and digital mixing and recording environments. These tools, as with notation software, are designed to manipulate and output a direct music representation. The representation, in this case, is MIDI note data, digital audio files, or sequences of event data. Again, new music is not created. The output is the direct representation that has been stored, edited, and processed by the user. Such systems often have modular processors (plug-ins or effects) for both MIDI and digital audio data. Some of these processors allow the user to control music data with indirect music representations. For example, a MIDI processor might implement an arpeggiator, letting the user, for a given base note, determine the scale, size, and movement of the arpeggio. In this case the configuration of the arpeggio is an indirect representation, and can be said to

have attributes of a CAAC system. This is not, however, the primary user-level interface.

2. RESEARCH IN CATEGORIZING COMPOSITION SYSTEMS

The number and diversity of CAAC systems, and the diversity of interfaces, platforms, and licenses, have made categorization elusive. Significant general overviews of computer music systems have been provided by Curtis Roads (1984, 1985), Loy and Curtis Abbott (1985), Bruce Pennycook (1985), Loy (1989), and Stephen Travis Pope (1993). These surveys, however, have not focused on generative or transformational systems.

Pennycook (1985) describes five types of computer music interfaces: (1) composition and synthesis languages, (2) graphics and score editing environments, (3) performance instruments, (4) digital audio processing tools, and (5) computer-aided instruction systems. This division does not attempt to isolate CAAC systems from tools used in modifying direct representations, such as score editing and digital audio processing. Loy (1989, p. 323) considers four types of languages: (1) languages used for music data input, (2) languages for editing music, (3) languages for specification of compositional algorithms, and (4) generative languages. This division likewise intermingles tools for direct representations (music data input and editing) with tools for indirect representations (compositional algorithms and generative languages). Pope's "behavioral taxonomy" (1993, p. 29), in focusing on how composers interact with software, is near to the goals of this study, but is likewise concerned with a much broader collection of systems, including "... software- and hardware-based systems for music description, processing, and composition ..." (1993, p. 26). Roads survey of "algorithmic composition systems" divides software systems into four categories: (1) self-contained automated composition programs, (2) command languages, (3) extensions to traditional programming languages, (4) and graphical or textual environments including music programming languages (1996, p. 821). This division also relates to the perspective taken here, though neither degrees of "self-containment" nor distinctions between music languages and language extensions are considered.

Texts that have attempted to provide an overview of CAAC systems in particular have generally used one of three modes of classification: (1) chronological (Hiller 1981; Burns 1994), (2) division by algorithm type (Dodge and Jerse 1997, p. 341; Miranda 2000), or (3) division by output format or output scale (Buxton 1978, p. 10; Laske 1981, p. 120). All of these methods, however, fail to isolate important attributes from the perspective of the user and developer. A chronological approach offers little information on similarities between historically disparate systems, and suggests,

incorrectly, that designs have developed along a linear trajectory. Many contemporary systems support numerous types of algorithms, and numerous types of output formats. This article proposes seven possible, and equally valid, descriptors of CAAC systems.

3. PRIMARY DESCRIPTORS

3.1. The Difficulty of Distinctions

Comparative software analysis is a difficult task, even if the software systems to be compared share a common purpose. Despite these challenges, such a comparison offers a useful vantage. Not only does a comparative framework demonstrate the diversity of systems available, it exposes similarities and relationships that might not otherwise be perceived.

In order to describe the landscape of software systems, it is necessary to establish distinctions. Rather than focusing on chronology, algorithms, or output types, this article proposes seven descriptors of CAAC system design. These descriptors are scale, process-time, idiom-affinity, extensibility, event production, sound source, and user environment. All systems can, in some fashion, be defined by these descriptors. For each descriptor, a range of specifications are given. These specifications, in some cases, represent a gradient. In all cases these specifications are non-exclusive: some systems may have aspects of more than one specification for a single descriptor. Importantly, all CAAC systems have some aspect of each descriptor. The use of multiple descriptors to describe a diverse field of systems is demonstrated by Pope in his “taxonomy of composer’s software” (1993), where eighteen different “dimensions” are proposed and accompanied by fifteen two-dimensional system graphs. Unlike the presentation here, however, some of Pope’s dimensions are only applicable to certain systems. John Biles, in his “tentative taxonomy” of evolutionary music systems (2003), likewise calls such descriptors “dimensions.”

It is unlikely that an objective method for deriving and applying a complete set of software descriptors is possible in any application domain, let alone in one that integrates with the creative process of music composition. Consideration of use case, technological change, and the nature of creative production requires broad categories with specifications that are neither mutually exclusive nor quantifiable. The assignment of specifications, further, is an interpretation open to alternatives. Though this framework is broad, its imprecision permits greater flexibility than previous attempts, while at the same time clearly isolating essential aspects of closely related systems from the entire history of the field.

3.2. Scale: Micro and Macro Structures

The scale of a CAAC system refers to the level of musical structures the system produces. Two extremes of a gradient are defined: micro and macro. Micro structures are musical event sequences commonly referred to as sound objects, gestures, textures, or phrases: small musical materials that require musical deployment in larger structures. Micro structures scale from the level of samples and grains to collections of note events. In contrast, macro structures are musical event sequences that approach complete musical works. Macro structures often articulate a musical form, such as a sonata or a chorale, and may be considered complete compositions. The concept of micro and macro structures closely relates to what Eduardo Reck Miranda (2000) calls bottom-up and top-down organizations, where bottom-up composition begins with micro structures, and top-down composition begins with macro structures.

Alternative time-scale labels for musical structures have been proposed. Horacio Vaggione has defined the lower limit of the macro-time domain as the note, while the micro-time domain is defined as sub-note durations on the order of milliseconds (2001, p. 60). Roads, in *Microsound* (2002, pp. 3-4), expands time into nine scales: infinite, supra, macro, meso, sound object, micro, sample, subsample, and infinitesimal. Macro, in the usage proposed here, refers to what Roads calls both macro and meso, while micro refers to what Roads calls meso, sound object, micro, and sample. Unlike the boundaries defined by Roads and Vaggione, the distinctions here are more fluid and highly dependent on context and musical deployment. Musical structure and temporal scales are, in part, a matter of interpretation. A composer may choose to create a piece from a single gesture, or to string together numerous large-scale forms.

Such a coarse distinction is useful for classifying the spectrum of possible outputs of CAAC systems. A few examples demonstrate the context-dependent nature of this descriptor. Xenakis’s *GENDYN*, for instance, is a system specialized toward the generation of micro structures: direct waveform break-points at the level of the sample. Although Xenakis used this system to compose entire pieces (*GENDY3* (1991), *S709* (1994)), the design of the software is specialized for micro structures. Though the system is used to generate music over a large time-span, there is little control over large-scale form (Hoffman 2000). Kemal Ebcioğlu’s *CHORAL* system (1988), at the other extreme, is a system designed to create a complete musical form: the Bach chorale. Though the system is used to generate music over a relatively short time-span, concepts of large-scale form are encoded in the system.

3.3. Process Model: Real-Time and Non-Real-Time

The process model of a CAAC system refers to the relationship between the computation of musical structures and their output. A real-time (RT) system outputs each event after generation along a scheduled time line. A non-real-time (NRT) system generates all events first, then provides output. In the context of a RT CAAC system, the calculation of an event must be completed before its scheduled output. Some systems offer a single process model while others offer both.

Whether a system is RT or NRT determines, to a certain extent, the types of operations that can be completed. RT processes are a subset of NRT processes: some processes that can be done in NRT cannot be done in RT. For example, a sequence of events cannot be reversed or rotated in RT (this would require knowledge of future events). Mikael Laurson, addressing the limitations of RT compositional processes, points out that a RT process model “can be problematic, or even harmful”: “composition is an activity that is typically ‘out-of-time’” and further, “there are many musical problems that cannot be solved in real time ... if we insist on real-time performance, we may have to simplify the musical result” (1996, p. 19). Though a CAAC system need not model traditional cognitive compositional activities (whether out-of-time or otherwise), a RT process model does enforce computational limits.

In general, a RT system is limited to linear processes: only one event, or a small segment of events (a buffer, a window, or a frame), can be processed at once. A NRT system is not limited to linear processes: both linear and nonlinear processing is available. A nonlinear process might create events in a sequential order different than their eventual output order. For example, event start times might be determined by a Gaussian distribution within defined time boundaries: the events will not be created in the order of their ultimate output. A RT system, however, has the obvious advantage of immediate interaction. This interaction may be in response to the composer or, in the case of an interactive music system, in response to other musicians or physical environments.

As with other distinctions, these boundaries are not rigid. A RT system might, instead of one event at a time, collect events into a frame and thus gain some of the functionality of NRT processing. Similarly, a NRT system, instead of calculating all events at once, might likewise calculate events in frames and then output these frames in RT, incurring a small delay but simulating RT performance.

Leland Smith’s SCORE system (1972), for example, has a NRT process model: music, motives, and probabilities are specified in a text file for each parameter, and this file is processed to produce a score. James McCartney’s SuperCollider language (1996) has a RT process model: with SuperCollider3 (SC3), instrument definitions

(SynthDefs) are instantiated as nodes on a server and respond to RT messages (McCartney 2002).

3.4. Idiom-Affinity: Singular and Plural

Idiom-affinity refers to the proximity of a system to a particular musical idiom, style, genre, or form. Idiom, an admittedly broad term, is used here to refer collectively to many associated terms. All CAAC systems, by incorporating some minimum of music-representation constructs, have an idiom-affinity. A system with a singular idiom-affinity specializes in the production of one idiom (or a small collection of related idioms), providing tools designed for the production of music in a certain form, from a specific time or region, or by a specific person or group. A system with a plural idiom-affinity allows the production of multiple musical styles, genres, or forms.

The idea of idiom-affinity is general. If a system offers only one procedural method of generating event lists, the system has a singular idiom-affinity. Idiom-affinity therefore relates not only to the design of low-level representations, but also to the flexibility of the large-scale music generators. The claim that all CAAC systems have an idiom-affinity has been affirmed by many researchers. Barry Truax states that, regardless of a system designer’s claims, “all computer music systems explicitly and implicitly embody a model of the musical process that may be inferred from the program and data structure of the system...” (1976, p. 230). The claim that all systems have an idiom-affinity challenges the goal of “musical neutrality,” a term used by Laurson to suggest that “the hands of the user should not be tied to some predefined way of thinking about music or to a certain musical style” (1996, p. 18). Laurson claims, contrary to the view stated here, that by creating primitives that have broad applicability and allowing for the creation of new primitives, a system can maintain musical neutrality despite the incorporation of “powerful tools for representing musical phenomena” (1996, p. 18). Musical neutrality can be approached, but it can never be fully realized.

Koenig’s PR1 (1970a), for example, is a system with a singular idiom-affinity: the system, designed primarily for personal use by Koenig, exposes few configurable options to the user and, in its earliest versions, offers the user no direct control over important musical parameters such as form and pitch. Paul Berg’s AC Toolbox (2003) has a plural idiom-affinity: low level tools and objects (such as data sections, masks, and stockpiles) are provided, but are very general, are not supplied with defaults, and can be deployed in a variety of configurations.

3.5. Extensibility: Closed and Open

Extensibility refers to the ability of a software system to be extended. This often means adding code, either in the

form of plug-ins or other modular software components. In terms of object-oriented systems, this is often done by creating a subclass of a system-defined object, inheriting low-level functionality and a system-compatible interface. An open system allows extensibility: new code can be added to the system by the user. A closed system does not allow the user to add code to the system or change its internal processing in any way other than the parameters exposed to the user.

In terms of CAAC systems, a relationship often exists between the extensibility of a system and its idiom-affinity. Systems that have a singular idiom-affinity tend to be closed; systems that have a plural idiom-affinity tend to be open. All open-source systems, by allowing users to manipulate system source code, have open extensibility. Closed-source systems may or may not provide open extensibility.

Joel Chadabe's and David Zicarelli's M (Zicarelli 1987; Chadabe 1997, p. 316), for instance, is a closed, stand-alone application: though highly configurable, new code, objects, or models cannot be added to the system or interface. Miller Puckette's cross-platform PureData (1997) is an open system: the language is open source and extensible through the addition of compiled modules programmed in C.

3.6. Event Production: Generation and Transformation

A distinction can be made between the generation of events from indirect music representations (such as algorithms or lists of musical materials) and the transformation of direct music representations (such as MIDI files) with indirect models. Within some CAAC systems, both processes are available, allowing the user to work with both the organization of generators and the configuration of transformers. Some systems, on the other hand, focus on one form over another. The division between generators and transformers, like other distinctions, is fluid and contextual.

Andre Bartetzki's Cmask system (1997) allows the generation of event parameters with a library of stochastic functions, generators, masks, and quantizers. Tools for transformation are not provided. Cope's EMI system (1996) employs a transformational model, producing new music based on analyzed MIDI files, extracting and transforming compositional patterns and signatures. Tools are not provided to generate events without relying on structures extracted from direct representations.

3.7. Sound Source: Internal, Exported, Imported, External

All CAAC systems produce event data for sound production. This event data can be realized by different sound sources. In some cases a system contains both the complete definition of sound-production components

(instrument algorithms), and is capable of internally producing the sound through an integrated signal processing engine. The user may have complete algorithmic control of not only event generation, but signal processing configuration. Such a system has an internal sound source. In other cases a system may export complete definitions of sound-production components (instrument algorithms) to another system. The user may have limited or complete control over signal processing configuration, but the actual processing is exported to an external system. For example, a system might export Csound instrument definitions or SuperCollider SynthDefs. Such a system has an exported sound source. In a related case a CAAC system may import sound source information from an external system, automatically performing necessary internal configurations. For example, loading instrument definitions into a synthesis system might automatically configure their availability and settings in a CAAC system. Such a system has an imported sound source. In the last case a system may define the sound source only with a label and a selection of sound-source parameters. The user has no control over the sound source except through values supplied to event parameters. Examples include a system that produces Western notation for performance by acoustic instruments, or a system that produces a Csound score for use with an external Csound orchestra. Such a system has an external sound source. As with other descriptors, some systems may allow for multiple specifications.

Roger Dannenberg's Nyquist (1997a, 1997b) is an example of a system with an internal sound source: the language provides a complete synthesis engine in addition to indirect music representations. The athenaCL system (Ariza 2005) is an example of a system that uses an exported sound source: a Csound orchestra file can be dynamically constructed and configured each time an event list is generated. Heinrich Taube's Common Music (1991) supports an imported sound source: Common Lisp Music (CLM) instruments, once loaded, are automatically registered within CM (1997, p. 30). Clarence Barlow's AutobusK system (1990) uses an external sound source: the system provides RT output for MIDI instruments.

3.8. User Environment: Language, Batch, Interactive

The user environment is the primary form in which a CAAC system exposes its abstractions to the user, and it is the framework in which the user configures these abstractions. A CAAC system may provide multiple environments, or allow users to construct their own environments and interfaces. The primary environment the system presents to the user can, however, be isolated.

Loy (1989, p. 319) attempts to distinguish languages, programs, and (operating) systems. Contemporary systems, however, are not so discrete: a "program" may allow internal scripting or external coding through the

program's API; a "language" may only run within a platform-specific program. Particularly in the case of CAAC systems, where minimal access to code-level interfaces is common, the division between "language" and "program" is not useful. Such features are here considered aspects of user environment. Language, batch, and interactive environments are isolated (and not discrete) because they involve different types of computer-user interaction. Loy even considers some systems, such as Koenig's PR1 and PR2, to be languages (1989, p. 324), even though, in the context of computer-user interaction, it has never been possible to program in the "language" of either system.

A language interface provides the user with an artificial language to design and configure music abstractions. There are two forms of languages: text and graphic. A text language is composed with standard text-editors, and includes programming languages, markup-languages, or formal languages and grammars. A graphic language (sometimes called a visual language) is used within a program that allows the organization of software components as visual entities, usually represented as a network of interconnected boxes upon a two-dimensional plane. A box may have a set of inputs and outputs; communication between boxes is configured by drawing graphic lines from inputs to outputs. Laurson (1996) provides a thorough comparison of text and graphic languages. He summarizes differences between the two paradigms: text languages offer compliance with standards, compactness, and speed, whereas graphic languages offer intuitive programming logic, intuitive syntax, defaults, and error checking (1996, p. 16). These differences are not true for all languages: some visual languages offer speed, while some text languages offer an intuitive syntax.

A batch interface is a system that only permits the user to provide input data, usually in the form of a text file or a list of command-line options. The input data, here called a manifest, is processed and the program returns a result. As Roads points out, batch processes refer "... to the earliest computer systems that ran one program at a time; there was no interaction with the machine besides submitting a deck of punched paper cards for execution and picking up the printed output" (1996, p. 845). Modern batch systems, in addition to being very fast, offer considerably greater flexibility of input representation. Though an old model, batch processing is still useful and, for some tasks, superior to interaction. The manifest may resemble a text programming language, but often lacks the expressive flexibility of a complete language. A batch system does not permit an interactive-session: input is processed and returned in one operation. What is desired from the software must be completely specified in the manifest. Curiously, Pope defines a batch system as distinct from RT and "rapid turnaround" systems not by its particular interface or user environment, but by "... the delay between the capture or description of signal, control, or event and its

audible effect" (1993, p. 29). More than just a performance constraint, modern batch environments define a particular form of user interaction independent of performance time or process model.

An interactive interface allows the user to issue commands and, for each command, get a response. Interactive interfaces usually run in a session environment: the user works inside the program, executing discrete commands and getting discrete responses. Interactive interfaces often have tools to help the user learn the system, either in the form of help messages, error messages, or user syntax correction. Interactive interfaces often let the user browse the materials that they are working with and the resources available in the system, and may provide numerous different representations of these materials. Such a system may be built with text or graphics. Focusing on interactive systems over interactive interfaces, Roads distinguishes between (1) "... light interactions experienced in a studio-based 'composing environment,' where there is time to edit and backtrack..." and (2) "... real-time interaction experienced in working with a performance system onstage, where ... there is no time for editing" (1996, p. 846). While this distinction is valuable for discussing context-based constraints of system use, many CAAC systems, with either language interfaces or interactive interfaces, support both types of system interaction as described by Roads. Here, use of interaction refers more to user-system interaction in NRT or RT production, rather than user-music interaction in RT production.

An interactive text interface is a program that takes input from the user as text, and provides text output. These systems often operate within a virtual terminal descended from the classic DEC VT05 (1975) and VT100 (1978) hardware. The UNIX shell is a common text interface. Contemporary text interfaces interact with the operating system and window manager, allowing a broad range of functionality including the production of graphics. These graphics, in most cases, are static and cannot be used to manipulate internal representations. An interactive text interface system may have a graphic user interface (GUI). Such a system, despite running in a graphic environment, conducts user interaction primarily with text. An interactive graphics interface employs a GUI for the configuration and arrangement of user-created entities. Users can alter musical representations by directly designing and manipulating graphics.

As with other descriptors, these specifications are not exclusive. A CAAC system may offer aspects of both a graphical and a textual programming language. The manifest syntax of a batch system may approach the flexibility of a complete text language. An interactive text or graphics system may offer batch processing or access to underlying system functionality as a language-based Application Programming Interface (API). Despite these overlapping environments, it is

nonetheless useful, when possible, to classify a system by its primary user-level interface.

William Schottstaedt's Pla system (1983) is an example of a text language. Laurson's Patchwork system (Laurson and Duthen 1989) provides an example of a graphical language. Mikel Kuehn's nGen (2001) is a batch user environment: the user creates a manifest, and this file is processed to produce Csound scores. Joel Chadabe's PLAY system demonstrates an interactive text interface, providing the user a shell-like environment for controlling the system (1978). Finally, Laurie Spiegel's Music Mouse system (1986) provides an example of an interactive graphic system.

4. ALGORITHMIC.NET

The definition and seven descriptors presented above are the result of extensive research in CAAC systems, much of which is beyond the scope of this article. This research has been made publicly available in the form of a website titled "algorithmic.net." This site provides a bibliography of over one thousand resources in CAAC and a listing of over eighty contemporary and historic software systems. For each system, references, links, descriptions, and specifications for the seven descriptors described above are provided. Flexible web-based tools allow users to search and filter systems and references, as well as to contribute or update information in the algorithmic.net database. The ultimate goal of this site is a collaborative lexicon of research in computer aided algorithmic music composition.

5. ACKNOWLEDGEMENTS

This research was funded in part by a grant from the United States Fulbright program, the Institute for International Education (IIE), and the Netherlands-America Foundation (NAF) for research at the Institute of Sonology, The Hague, the Netherlands. Thanks to Paul Berg and Elizabeth Hoffman for commenting on earlier versions of this article, and to the ICMC's anonymous reviewers for valuable commentary and criticism.

6. REFERENCES

Ariza, C. 2005. *An Open Design for Computer-Aided Algorithmic Music Composition: athenaCL*. Ph.D. Dissertation, New York University.

Barlow, C. 1990. "Autobus: An algorithmic real-time pitch and rhythm improvisation programme." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 166-168.

Bartzki, A. 1997. "CMask, a Stochastic Event Generator for Csound." Internet: <http://gigant.kgw.tu-berlin.de/~abart/CMaskMan/CMask-Manual.htm>.

Berg, P. and R. Rowe, D. Theriault. 1980. "SSP and Sound Description." *Computer Music Journal* 4(1): 25-35.

Berg, P. 2003. *Using the AC Toolbox*. Den Haag: Institute of Sonology, Royal Conservatory.

Biles, J. A. 2003. "GenJam in Perspective: A Tentative Taxonomy for GA Music and Art Systems." *Leonardo* 36(1): 43-45.

Burns, K. H. 1994. *The History and Development of Algorithms in Music Composition, 1957-1993*. D.A. Dissertation, Ball State University.

Chadabe, J. 1978. "An Introduction to the Play Program." *Computer Music Journal* 2(1).

———. 1997. *Electric Sound: The Past and Promise of Electronic Music*. New Jersey: Prentice-Hall.

Cope, D. 1991. *Computers and Musical Style*. Oxford: Oxford University Press.

———. 1993. "Algorithmic Composition [re]Defined." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 23-25.

———. 1996. *Experiments in Music Intelligence*. Madison, WI: A-R Editions.

Dannenberg, R. B. 1997a. "The Implementation of Nyquist, A Sound Synthesis Language." *Computer Music Journal* 21(3): 71-82.

———. 1997b. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis." *Computer Music Journal* 21(3): 50-60.

Dodge, C. and T. A. Jerse. 1997. *Computer Music; Synthesis, Composition, and Performance*. Wadsworth Publishing Company.

Ebcioğlu, K. 1988. "An Expert System for Harmonizing Four-part Chorales." *Computer Music Journal* 12(3): 43-51.

Gerhard, D. and D. H. Hepting. 2004. "Cross-Modal Parametric Composition." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 505-512.

Hiller, L. 1981. "Composing with Computers: A Progress Report." *Computer Music Journal* 5(4).

Hiller, L. and L. Isaacson. 1959. *Experimental Music*. New York: McGraw-Hill.

Hoffman, P. 2000. "A New GENDYN Program." *Computer Music Journal* 24(2): 31-38.

Koenig, G. M. 1970a. "Project One." In *Electronic Music Report*. Utrecht: Institute of Sonology. 2: 32-46.

- . 1970b. "Project Two - A Programme for Musical Composition." In *Electronic Music Report*. Utrecht: Institute of Sonology. 3.
- Kuehn, M. 2001. "The nGen Manual." Internet: <http://mustec.bgsu.edu/~mkuehn/ngen/man/ngenman.htm>.
- Laske, O. 1981. "Composition Theory in Koenig's Project One and Project Two." *Computer Music Journal* 5(4).
- Laurson, M. and J. Duthen. 1989. "PatchWork, a Graphical Language in PreForm." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 172-173.
- Laurson, M. 1996. *Patchwork*. Helsinki: Sibelius Academy.
- Loy, D. G. 1989. "Composing with Computers: a Survey of Some Compositional Formalisms and Music Programming Languages." In *Current Directions in Computer Music Research*. M. V. Mathews and J. R. Pierce, eds. Cambridge: MIT Press. 291-396.
- Loy, D. G. and C. Abbott. 1985. "Programming Languages for Computer Music Synthesis, Performance, and Composition." *ACM Computing Surveys* 17(2).
- Marino, G. and M. Serra, J. Raczinski. 1993. "The UPIC System: Origins and Innovations." *Perspectives of New Music* 31(1): 258-269.
- Mathews, M. V. and L. Rosler. 1968. "Graphical Language for the Scores of Computer-Generated Sounds." *Perspectives of New Music* 6(2): 92-118.
- McCartney, J. 1996. "SuperCollider: a New Real Time Synthesis Language." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association.
- . 2002. "Rethinking the Computer Music Language." *Computer Music Journal* 26(4): 61-68.
- Miranda, E. R. 2000. *Composing Music With Computers*. Burlington: Focal Press.
- Pennycook, B. W. 1985. "Computer Music Interfaces: A Survey." In *ACM Computing Surveys*. New York: ACM Press. 17(2): 267-289.
- Pope, S. T. 1993. "Music Composition and Editing by Computer." In *Music Processing*. G. Haus, ed. Oxford: Oxford University Press. 25-72.
- Puckette, M. 1997. "Pure Data." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 224-227.
- Roads, C. 1984. "An Overview of Music Representations." In *Musical Grammars and Computer Analysis*. Firenze: Leo S. Olschki. 7-37.
- . 1985. "Research in music and artificial intelligence." In *ACM Computing Surveys*. New York: ACM Press. 17(2): 163-190.
- . 1996. *The Computer Music Tutorial*. Cambridge: MIT Press.
- . 2002. *Microsound*. Cambridge: MIT Press.
- Schottstaedt, W. 1983. "Pla: A Composer's Idea of a Language." *Computer Music Journal* 7(1).
- Simoni, M. 2003. *Algorithmic Composition: A Gentle Introduction to Music Composition Using Common LISP and Common Music*. Ann Arbor: Scholarly Publishing Office, the University of Michigan University Library.
- Smith, L. 1972. "SCORE - A Musician's Approach to Computer Music." *Journal of the Audio Engineering Society* 20(1): 7-14.
- Spiegel, L. 1986. "Music Mouse: An Intelligent Instrument." Internet: <http://retinary.org/ls/programs.html>.
- . 1989. "Distinguishing Random, Algorithmic, and Intelligent Music." Internet: http://retinary.org/ls/writings/alg_comp_ltr_to_cem.html.
- Supper, M. 2001. "A Few Remarks on Algorithmic Composition." *Computer Music Journal* 25(1): 48-53.
- Taube, H. 1991. "Common Music: A Music Composition Language in Common Lisp and CLOS." *Computer Music Journal* 15(2): 21-32.
- . 1997. "An Introduction to Common Music." *Computer Music Journal* 21(1): 29-34.
- . 2004. *Notes from the Metalevel: An Introduction to Computer Composition*. Swets Zeitlinger Publishing.
- Truax, B. 1976. "A Communicational Approach to Computer Sound Programs." *Journal of Music Theory* 20(2): 227-300.
- Vaggione, H. 2001. "Some Ontological Remarks about Music Composition Processes." *Computer Music Journal* 25(1): 54-61.
- Xenakis, I. 1965. "Free Stochastic Music from the Computer. Programme of Stochastic music in Fortran." *Gravesaner Blätter* 26.
- . 1992. *Formalized Music: Thought and Mathematics in Music*. Indiana: Indiana University Press.
- Zicarelli, D. 1987. "M and Jam Factory." *Computer Music Journal* 11(4): 13-29.