An Architectural Framework
for Interactive Music Systems

Alexandre R.J. François and Elaine Chew
Viterbi School of Engineering
University of Southern California
Los Angeles, CA, USA

{afrancoi,echew}@usc.edu

ABSTRACT

This paper introduces the Software Architecture for Immersipresence (SAI) framework to the computer music community. SAI is a software architecture model for designing, analyzing and implementing applications that perform distributed, asynchronous parallel processing of generic data streams. The most significant innovation of SAI is its ability to handle real-time DSP, interactive control, and data-centered representations in a unified model. This generality facilitates the design and implementation of complex interactive systems that combine music analysis, synthesis and on-line control. Two examples illustrate the use of SAI in the design and implementation of interactive music systems: MuSA.RT, a system for real-time analysis and interactive visualization of tonal patterns in music, and ESP, a driving interface (wheel, pedals and display) for creating expressive performances from expressionless music files.

Keywords

Software Architecture, Interactive Systems, Music software

1. INTRODUCTION

This paper introduces the Software Architecture for Immersipresence (SAI) framework [15] to the computer music community. SAI is a software architecture model for designing, analyzing and implementing applications that perform distributed, asynchronous parallel processing of generic data streams. Two examples illustrate the use of SAI in the design and implementation of interactive music systems.

The design principles underlying SAI are rooted in François' past and ongoing cross-disciplinary research on interactive systems at the Integrated Media Systems Center (IMSC), an NSF Engineering Research Center at the University of Southern California, and on computer vision systems, at USC's Institute for Robotics and Intelligent Systems. Although originally motivated by the Immersipresence vision–that is, combining immersion and interactivity [20]–, SAI provides a general formalism for the design,

analysis and implementation of *complex software systems of asynchronous interacting processing components.*

SAI defines architectural level abstractions that are consistent with a general model of computation. These abstractions resolve a class of seemingly related fundamental issues, that are characerized by Puckette as a divide between processing and representation paradigms [26], and by Dannenberg as the difficulty of combining (functional) signal processing and (imperative) event processing [9].

Interactive systems are particularly interesting and challenging, as they require on-line (real-time) analysis and synthesis of data media of different nature. SAI is designed explicitly to address the limitations of traditional approaches in this context. For historical reasons, SAI was first applied to the design and implementation of real-time and interactive computer vision systems [17]. Its relevance in the context of interactive music systems was explored and established over the past few years through collaborations between the authors, started at IMSC.

The remainder of this paper is organized as follows. Section 2 relates the SAI approach to major landmarks in the rich computer music history. Section 3 describes the main features of the framework: the architectural abstractions that form the SAI style, important properties, and design and implementation tools. Section 4 demonstrates the use of the framework with two interactive music systems. Finally, Section 5 offers concluding remarks and outlines research and development directions for ongoing and future work.

2. RELATED WORK

Amatriain's thorough review and analysis of the audio and music processing software landscape in his thesis [1] testifies to the richness and diversity of this field.

Existing approaches can be characterized according to various criteria. For example, levels of abstraction range from programming languages to code libraries, to application frameworks, to programming environments (possibly visual), and graphical applications. Another criterion is the primary objective, which can be music or audio analysis, synthesis, or composition. Requirements may include real-time constraints or interactivity (i.e. low latency).

Focusing on interactive systems and their underlying models of computation separates *on-line* from *off-line* approaches. In the on-line group, synthesis-oriented efforts adopt models from Digital Signal Processing (DSP), and are often concerned with scheduling (because of the hard real-time requirement). This category comprises of the lineage rooted in the Music N languages [18], which includes Csound [28, 27, 13], the Max paradigm [25] in its various forms (e.g. Max/MSP [19] and Pure Data [24]), and oth-

ers. All of these assume a process oriented dataflow model, which is not adapted to interactive manipulation. Researchers have introduced various mechanisms for interactive control in on-line DSP-style systems (either synthesis- or analysis- oriented). For example, in Aura [9] a message-passing model complements the dataflow model. This hybridation occurs at a rather low level of abstraction (language).

The off-line group comprises of analysis and composition oriented systems that focus on representation. The interactive nature of these systems, if any, comes from the existence of a visual graphical user interface (GUI) to manipulate the structures or their computations (note that GUIs are typically built on message passing models). OpenMusic [22, 3, 4] is a representative member of this category. Its visual interface adopts the popular patch metaphor used in Max/MSP and most visual environments for music processing, but it does so with very different semantics: as observed by Puckette in [26], Max patches contain dynamic process information, while OpenMusic patches contain static data. The dataflow model of OpenMusic maps to the functional programming approach, and has no model for representing or handling real-time (or on-line) events.

Both semantics are useful for different purposes, and part of their appeal is that they define high-level abstractions. Unfortunately, the two models are incompatible. Assayag and Dubnov's improvisation system [2] provides a concrete example. Off-line versions of the learning and generation algorithms are implemented in OpenMusic. The on-line implementation of the system, OMax [11], utilizes the OpenMusic implementation in conjunction with Max/MSP to handle the on-line aspects, such as real-time control, MIDI and audio acquisition and rendering. Communication between, and coordination of, the two subsystems requires the use of a special interaction protocol, OpenSound Control [23].

SAI introduces abstractions that aim to reconcile real-time DSP, interactive control, and data-centered representations in a unified model. SAI can therefore be seen as a hybrid architectural style (the approach taken here), or as a more general model in which the ones listed above can be characterized as *architectural patterns*.

SAI's abstractions are high-level (architectural) ones that are independent of the programming models used to implement them. SAI is therefore different in nature from programming and application frameworks such as CLAM [1]. (The relationship between SAI and code libraries will be addressed below.) SAI abstractions are also independent of any visual metaphors that might be employed to manipulate them.

3. THE SAI FRAMEWORK

SAI constitutes a framework in the sense that it defines a set of concepts and abstractions that together constitute a model of some application domain. The model can be instantiated to represent a particular application. In an attempt to providing a unifying model, the application domain targeted for SAI is, by choice, extremely general. The SAI model builds on three pillars: (1) an explicit account of time both in data and processing models; (2) the distinction between persistent and volatile data; and, (3) asynchronous parallelism. This section describes SAI's abstractions, formalized as an *architectural style*; a few interesting properties of the style; and, existing design and implementation tools.
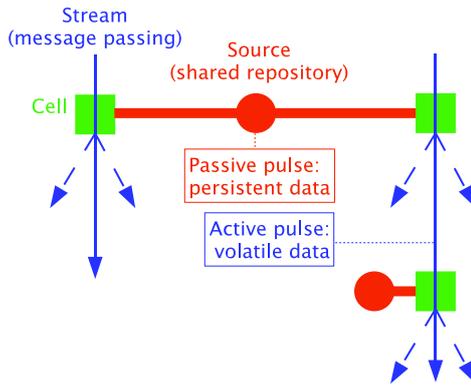

Figure 1: SAI elements and graphical notation.

3.1 Architectural style

SAI specifies a formal architectural style [16] comprising of an extensible data model and a hybrid (shared memory and message-passing) distributed asynchronous parallel processing model. Figure 1 presents an overview of SAI defining elements in their standard graphical notation.
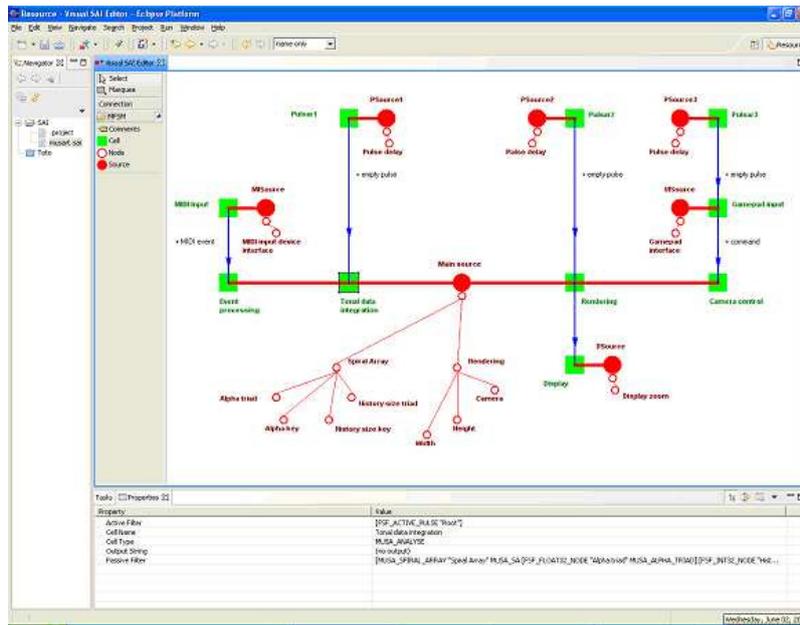
In SAI, all data is encapsulated in *pulses*. A pulse is the carrier for all the synchronous data corresponding to a given time stamp. Information in a pulse is organized as a mono-rooted composition hierarchy of *node* instances. The nodes constitute an extensible set of atomic data units that implement or encapsulate specific data structures. Pulses holding volatile data flow down streams defined by connections between processing centers called *cells*, in a message passing fashion. They trigger computations, and are thus called *active* pulses. In contrast, pulses holding persistent information are held in repositories called *sources*, where the processing centers can access them in a concurrent shared memory access fashion. Processing in a cell may result in the augmentation of the active pulse (input data), and augmentation and/or update of the passive pulse (process parameters). The processing of active pulses is carried in parallel, as they are received by the cell. Data binding is performed dynamically in an operation called *filtering*. Active and passive *filters* qualitatively specify, for each cell, the target data in respective pulses. This hybrid model combining message passing and shared repository communication, combined with a unified data model, constitutes a universal processing framework.

A particular system architecture is specified at the conceptual level by a set of source and cell instances, and their inter-connections. Specialized cells may be accompanied by a description of the task they implement. The logical level specification of a design describes, for each cell, its active and passive filters and its output structure, and for each source, the structure of its passive pulse.

In the graphical notation, cells are represented as squares, sources as circles. Source-cell connections are drawn as fat lines, while cell-cell connections are drawn as thin arrows crossing over the cells. When color is available, cells are colored in green (reserved for processing); sources, source-cell connections, and passive pulses are in colored in red (persistent information); and, streams and active pulses are colored in blue (volatile information).

3.2 Architectural properties

By design, the SAI style shares many of the desirable

Figure 2: VisualSAI: a prototype Integrated Visual Architecture Design and Analysis Environment. The graph shown is that of the MuSA.RT system.

properties of classical dataflow models. The explicit representation of the flow of data allows for the intuitive design and fast high level understanding of a system's components and their interactions. The modularity of the model facilitate distributed development and testing of particular elements, and easy maintenance and evolution of existing systems. SAI also naturally supports distributed and parallel processing. The SAI style provides unified data and processing models for generic data streams, allowing for the simultaneous modeling of DSP, control, and data-centered approaches.

The underlying asynchronous parallel processing model promotes designing for optimal (theoretical) system latency and throughput. SAI abstractions do not impose arbitrary hard real-time constraints, but rather suggest (and enable) the implementation of synchronization mechanisms *only* when needed. On current hardware, most interactive systems do not require hard synchronization, but rather best-effort performance for low-latency, complemented by time consistency checks. Modeling systems as inherently dynamic, and with explicit account of real-time, leads to the natural expression and efficient implementation of such requirements.

The distinction between volatile and persistent data is key to providing a consistent model for both process- and data-centered approaches. The resource and instance models described in [10] illustrates the collapse of these two notions in classical models: the instrument is a persistent model while a note generated by the instrument is a volatile message. The examples described in the next section offer other illustrations of persistent and volatile data.

SAI spawns a continuum of intermediate-level representations from conceptual to physical specifications, a property reinforced by the graphical notation. SAI promotes the encoding of system logic in the structural organization of simple computing components rather than in the complexity of the computations carried by individual compo-

nents. SAI designs exhibit a rich variety of structural and functional architectural patterns, whose systematic study will produce tools for assisting in design and re-use.

3.3 Design and implementation tools

An open source architectural middleware called Modular Flow Scheduling Framework (MFSM) [21], developed in C++, provides cross-platform code support for SAI's architectural abstractions, in the form of an extensible set of classes. MFSM is (heavily) multi-threaded and transparently leverages hyper-threading and multiprocessor architectures. A number of software modules regroup specializations that implement specific data structures, algorithms and/or functionalities. They constitute a constantly growing base of open source, reusable code, maintained as part of the MFSM project. In particular, the MFSM framework facilitates the leveraging of existing third-party code libraries through encapsulation. Extensive documentation, including a user guide, a reference guide, and tutorials, are available on the project Web site.

The graphical and compositional nature of the SAI model suggests the creation of integrated visual design and analysis environments. The formal nature of the model make it suitable for automatic static analysis and code generation (currently under development).

Figure 2 shows a screen shot of VisualSAI [29], a prototype of the user interface for such an environment, implemented as a plug-in for the Eclipse platform [12]. (The SAI graph shown is that of the MuSA.RT system described below.) As already noted, the SAI model is not linked to any visual metaphor, so that custom environments dedicated to specific activities are possible.

4. EXAMPLE SYSTEMS

This section describes two examples of interactive music systems designed using the SAI model. Both exhibit architectural patterns typical of interaction [14]. They are implemented using the MFSM middleware and share a sig-
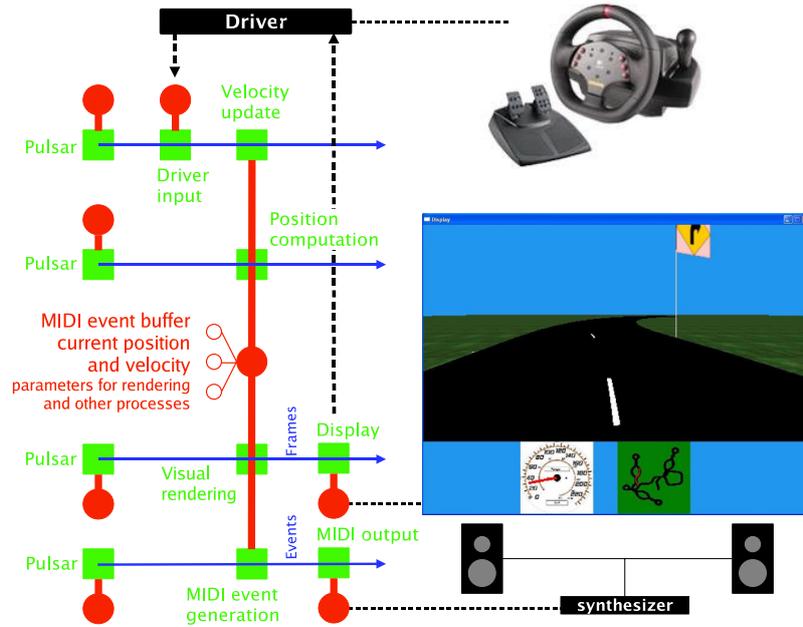
Figure 3: MuSA.RT synopsis with conceptual level system architecture in SAI notation.

nificant amount of code that exist in the form of MFSM modules.

4.1 The MuSA.RT System

MuSA.RT (Music on the Spiral Array . Real-Time) [7, 6] is a system for real-time analysis and interactive visualization of tonal patterns in music. Figure 3 presents a synopsis of the project with a conceptual level system architecture in SAI notation.

The system processes and analyzes MIDI input, for example captured during a live performance, in real-time, and maps the data to the Spiral Array [5], a 3D model for tonality. A center of effect (CE) summarizes contextual tonal information by mapping any pitch collection to a spatial point, and any time series of notes to meaningful trajectories, inside the Spiral Array. CE trajectory analysis allows the viewer to infer the presently active set of pitch classes, and higher level constructs, such as the current chord and key, revealed through real-time 3D rendering. An operator can concurrently navigate through the Spiral Array space using a gamepad to zoom in and out, tilt the viewing angle and circle around the spiral to get a better view of the tonal structures. An automatic pilot option seeks the best view angle and centers the camera at the heart of the action.

The system therefore combines on-line analysis of music data, computation of dynamic music structures, and real-time synthesis of visual data with interactive parameter adjustments. These computations are defined by four independent streams: (1) MIDI input and event processing; (2) tonal analysis (real-time CE algorithms); (3) rendering of the Spiral Array structures; and, (4) control device (gamepad) input and camera manipulation. These four streams potentially operate according to different modalities (e.g. push or pull input models) and at different rates. The Spiral Array structure, processing and rendering parameters are persistent (yet dynamic) data; the MIDI mes-

sages and the rendered frames for visualization are volatile data.

The precise scheduling and synchronization of multiple data streams processed and synthesized in real-time would constitute a major challenge in creating the MuSA.RT system adopting a traditional approach. Instead, the SAI model provides the tools to design an architecture that ensures best achievable latency between input and visual feed-back to the spectators.

From an engineering point-of-view, the complexity of such cross-disciplinary experiments is traditionally limited by actual system integration, which is the main source of unforeseen problems. Using SAI and MFSM greatly simplified system design, implementation and integration. From research point-of-view, the MuSA.RT system constitutes a platform for testing and validating the different modules involved. Each functional module can be replaced by a functionally equivalent module, allowing strictly controlled comparisons in an otherwise identical setting.

4.2 The ESP System

ESP(The Expression Synthesis Project) [8] is a driving interface (wheel, pedals and display) for creating expressive performances from expressionless music files. The use of a compelling and intuitive metaphor makes high-level expressive decisions accessible to non-experts. Figure 4 presents a synopsis of the project with a conceptual level system architecture in SAI notation.

The performer drives the car through a road in a virtual world. The driving (specifically, the state of the dynamic car model) directly controls the tempo and loudness of the music as it unfolds over time. The turns in the road are based on music structure, and guide the user's expressive choice.

The ESP system combines analysis of user input, on-line computations of the dynamic model, and real-time synthesis of synchronous visual and music data. These computa-

Figure 4: ESP synopsis with conceptual level system architecture in SAI notation.

tions are defined by four independent streams: (1) control device (driving wheel and pedals) input and velocity update for the car model; (2) discrete time integration of the dynamic equations for the car model; (3) visual rendering of the driver's view (including dashboard instruments); and, (4) aural rendering (MIDI event generation).

The architecture of the ESP system exhibits similarities in structure and function (patterns) with that of the MuSA.RT system. The score (MIDI event buffer), car model, processing and rendering parameters are persistent (yet dynamic) data; the MIDI messages and the frames rendered for visualization are volatile data.

The ESP system does not explicitly implement any hard synchronization mechanism to ensure that the music synthesized and visual data are presented with some fixed temporal error bound. Although by no means ruled out by the SAI model, such a mechanism would be computationally expensive, and difficult to certify, given the complexity of the network of external software and hardware elements involved in the final result. Rather, the design minimizes latency on each stream. "Synchronous effect" is achieved when the time discrepancies are below the threshold of human perception, a performance level usually achievable on modern commodity computing platforms.

5. SUMMARY AND FUTURE WORK

This paper introduced the SAI framework in the context of computer music systems. SAI defines high-level abstractions that form a general formalism for the design, analysis and implementation of complex software systems of asynchronous interacting processing components. The open source architectural middleware MFSM provides an extensible set of classes that implement SAI's architectural abstractions.

The most significant innovation of the SAI framework is its ability to handle real-time DSP, interactive control, and data-centered representations in a unified model. This generality facilitates the design and implementation of complex interactive systems that combine music analysis, synthesis, and on-line control. For example, Figure 5 shows a possible architecture for a real-time improvisation system such as that of Assayag and Dubnov [2]. The development of audio- and music-oriented code modules, especially modules encapsulating existing libraries, will help motivate the use of the framework for music system design and implementation.

The SAI framework also aims to facilitate the design and development of interactive music systems that involve other input and output modalities, such as graphics and vision. The collection of available modules pertaining to these domains is more substantial and constantly growing (see for example the open source WebCam Computer Vision project [30]).

As research on theoretical aspects of SAI and tool development continue, it is the adoption of the framework in various interacting fields of research that bears the promise of ground-breaking cross-disciplinary explorations.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a Focus on Audio and Music.* PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, 2004.
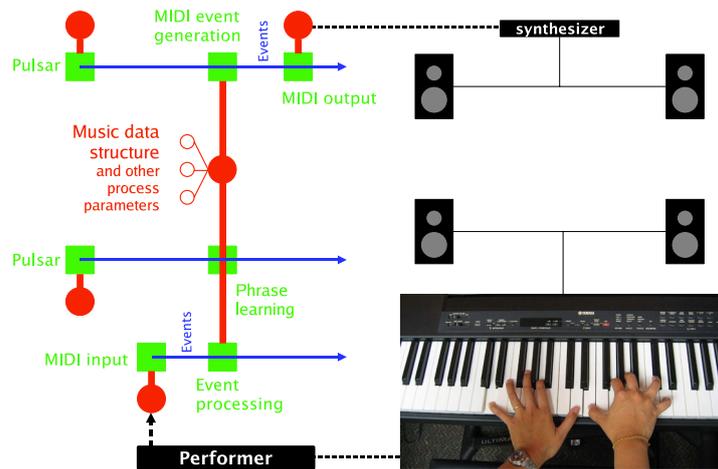
Figure 5: Possible architecture for an improvisation system.

[2] G. Assayag and S. Dubnov. Using factor oracles for machine improvisation. *Soft Computing*, 8:1–7, 2004.

[3] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue. Computer assisted composition at IRCAM: PatchWork & OpenMusic. *Computer Music Journal*, 23(3), 1999.

[4] J. Bresson, C. Agon, and G. Assayag. OpenMusic 5: A cross-platform release of the computer-assisted composition environment. In *Proc. 10th Brazilian Symposium on Computer Music*, Belo Horizonte, Brazil, 2005.

[5] E. Chew. *Towards a Mathematical Model of Tonality*. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000.

[6] E. Chew and A. R. François. MuSA.RT - Music on the Spiral Array . Real-Time. In *Proceedings of ACM Multimedia*, Berkeley, CA, USA, November 2-8 2003.

[7] E. Chew and A. R. François. Interactive multi-scale visualizations of tonal evolution in MuSA.RT opus 2. *ACM Computers in Entertainment*, 2(4), October-December 2005.

[8] E. Chew, A. R. François, J. Liu, and A. Yang. ESP: A Driving Interface for Expression Synthesis. In *Proc. New Instruments for Musical Expression*, Vancouver, B.C., Canada, May 2005.

[9] R. B. Dannenberg. A language for interactive audio applications. In *Proc. International Computer Music Conference*, San Francisco, CA, 2002.

[10] R. B. Dannenberg, D. Rubine, and T. Neuendorffer. The resource-instance model of music representation. In *Proc. International Computer Music Conference*, Montreal, Quebec, Canada, 1991.

[11] S. Dubnov and G. Assayag. Improvisation planning and jam session design using concepts of sequence variation and flow experience. In *Proc. International Conference on Sound and Music Computing*, Salerno, Italy, November 2005.

[12] Eclipse. *www.eclipse.org*.

[13] R. B. (Ed.). *The Csound Book*. MIT Press, Cambridge, MA, 2000.

[14] A. R. François. Components for immersion. In *Proc. IEEE International Conference on Multimedia and Expo*, Lausanne, Switzerland, August 2002.

[15] A. R. François. Software architecture for immersipresence. Technical Report IMSC-03-001, Integrated Media Systems Center, University of Southern California, December 2003.

[16] A. R. François. A hybrid architectural style for distributed parallel processing of generic data streams. In *Proc. International Conference on Software Engineering*, pages 367–376, Edinburgh, Scotland, UK, May 2004.

[17] A. R. François. Software Architecture for Computer Vision. In G. Medioni and S. Kang, editors, *Emerging Topics in Computer Vision*, pages 585–654. Prentice Hall, Upper Saddle River, NJ, 2004.

[18] M. Matthews. *The Technology of Computer Music*. MIT Press, Cambridge, MA, 1969.

[19] Max/MSP. *www.cycling74.com*.

[20] D. McLeod, U. Neumann, C. Nikias, and A. Sawchuk. Integrated media systems. *IEEE Signal Processing Magazine*, 16(1):33–43, Jan. 1999.

[21] Modular Flow Scheduling Middleware. *mfsm.sourceForge.net*.

[22] OpenMusic. *recherche.ircam.fr/equipes/repmus/OpenMusic*.

[23] OpenSound Control. *www.cnmat.berkeley.edu/OpenSoundControl*.

[24] M. S. Puckette. Pure data: another integrated computer music environment. In *Proc. Second Intercollege Computer Music Concerts*, pages 37–41, Tachikawa, Japan, 1996.

[25] M. S. Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

[26] M. S. Puckette. A divide between 'compositional' and 'performative' aspects of Pd. In *Proc. First International Pd Convention*, Graz, Austria, 2004.

[27] B. L. Vercoe. Extended Csound. In *Proc. International Computer Music Conference*, pages 141–142, Hong Kong, China, 1996.

[28] B. L. Vercoe and D. P. Ellis. Real-time Csound: software synthesis with sensing and control. In *Proc. International Computer Music Conference*, pages 209–211, Glasgow, Scotland, 1990.

[29] VisualSAI. *visualsai.sourceForge.net*.

[30] WebCam Computer Vision. *wccv.sourceForge.net*.