A Protocol for creating Multiagent Systems in Ensemble with Pure Data

Pedro Bruel
Universidade de São Paulo
pedro.bruel@gmail.com

Marcelo Queiroz
Universidade de São Paulo
mqz@ime.usp.br

ABSTRACT

This work presents a protocol for integration of two software platforms, the Ensemble framework for musical multiagent systems and the Pure Data programming environment. Ensemble is written in Java and requires knowledge of this language in order to access high-level features of the framework, such as creating customized agent reasonings, new event servers for non-supported data-types, or new physical models for the virtual world. On the other hand, Pure Data (Pd) is a very popular programming environment for real-time audio processing (among other things) and has an ever-growing community of users interested in sound and music applications. The protocol described here allows Pd users with no knowledge of Java to create musical multiagent applications in Ensemble with a high degree of flexibility, including configuration of parameters defining the virtual world, creation of agents and agent components (sensors, actuators, memories and knowledge base) and the definition of agent reasonings, which control agent behaviour and agent interactions in the virtual world, all from within Pd patches.

1. INTRODUCTION

The interest in multiagent systems in music started about fifteen years ago, and several such applications have appeared in the literature, which included distributed artificial intelligence concepts such as autonomous agents, virtual world modelling and collaborative agent interactions, in musical environments dealing with composition, improvisation and performance [1, 2, 3, 4, 5, 6, 7, 8]. Many of these applications [9, 10, 11, 12, 13, 14, 15, 16] presented a whole conception of the virtual world and its laws, and of agents with specific cognitive and musical functions, following determined algorithms.

A few exceptions to this rule are systems designed to aid the development of general musical multiagent applications, such as MAMA [4], ISO [5] and Ensemble [6, 8]. MAMA focused in agents which exchanged and produced exclusively MIDI information; ISO is oriented towards the idea of a swarm orchestra, where agents in a swarm are controlled by complex collective behaviours. Ensemble on the other hand concentrates on the idea of autonomous agents, and has tackled from its inception both symbolic exchange (MIDI, text, musical data and algorithms) and audio communication between agents immersed in a virtual world, which required physical modelling of sound propagation in various virtual realities (including, but not restricted to, realistic 3D spherical sound propagation).

The Ensemble framework has been written in Java, to allow some degree of platform-independence, uses the JADE multiagent middleware and depends on a sound engine, such as Java Sound (over the native OS sound server), PortAudio or JACK. Although many existing components in Java may be combined and assembled through XML configuration files to produce a number of musical multiagent applications, the high degree of flexibility that the framework offers is only available to a Java-literate user, who is thus able to design new components, new reasonings, new event types and event servers, and ultimately new virtual realities. Unfortunately for Ensemble, a number of potentially interested users are not Java programmers and their interest wane before the perspective of having to dive into Java code.

libpd [17] is a project which allows programmers to access and control Pure Data objects, patches and the whole DSP engine from within other applications. The libpd API is written in C and has language bindings for Java, which allows Ensemble to benefit from this infrastructure, allowing parts of the framework, that were originally meant to be written in Java, to be defined in Pd and accessed through libpd's API. On one hand, this possibility requires not only the incorporation (in Ensemble) of mechanisms for accessing and controlling Pd patches and the data that flows between Ensemble and these patches, but also defining a complete protocol for accessing Ensemble agent structures, such as sensors, actuators, memories and its knowledge base, from within Pd patches. On the other hand, this integration provides the user with the ability of completely defining high level agent processes through Pd patches, and unleashes Ensemble from its requirement of Java-literacy (on the part of the user).

This paper is structured as follows. Section 2 presents the structures in Ensemble dealing with agent reasoning, including components that exchange information with a reasoning, such as sensors and actuators, memories and knowledge bases. Section 3 describes libpd and the functions that allow control of Pd patches from within Ensemble. Section 4 describes the implemented Pd-Ensemble protocol, which allow access to Ensemble structures from within a Pd patch. Section 5 shows examples of how to use the protocol to configure and control Ensemble applications. To conclude, section 6 evaluates the contributions

of the proposed implementation and discusses the remaining aspects of Ensemble that still require Java knowledge, which lead to proposals of future work.

2. AGENT REASONING IN ENSEMBLE

Musical agents in Ensemble are computational entities inhabiting a virtual world, with specific physical and biological laws which are defined by the user (the application designer). They use sensors and actuators as mediators for all interactions with the virtual world and with other agents. For instance, hearing sensors and sound-producing actuators are used for capturing sound from the environment and producing sound in the environment, which will reach other agents according to a user-defined sound propagation law; motion actuators are used to allow the agent to control its position within the virtual world, and contact and sight sensors can be used to grasp information about external agents and objects [6].

The environment itself is an agent in Ensemble, which controls all kinds of agent-environment and agent-agent interactions through event servers dealing with specific types of information. For instance, an audio event server is the recipient of all sound events produced by each (sound-producing) actuator in the system, and is responsible for carrying out the prescribed sound propagation law in order to deliver sound to each (sound-capturing) sensor. Sensors and actuators dealing with each type of information are required to register themselves with the corresponding event server in order to guarantee proper working of the framework. Many pre-defined sensors and actuators for most common tasks, such as sound-capturing and sound-producing, motion and text message exchange, are available with all the technical details already implemented and can be used readily in applications [8].

Reasonings play a central role in agent interactions, because in these components are the cognitive mechanisms that analyse, combine, decide and synthesize every action and sound that the agent is going to produce in the virtual environment. Although creating agents and plugging components can be done in Ensemble through simplified XML initialization files, defining new forms of complex behaviour for Ensemble agents was only possible by writing new Java components within the framework.

Reasonings are able to operate on several data that the agent has access to, including sensor/actuator values and the knowledge base, which is used to hold facts, theories, memories, etc. Memories are a specialized type of stream data that the agent may instantiate and use for its algorithmic processes, but are also automatically handled in the case of sensor/actuator stream values. A sensor has a corresponding read-only memory that holds past received data and is continually updated with the sensor's input, and an actuator reads the data that it will produce from a corresponding memory, which has been previously filled by the reasoning that wants to control that actuator.

In Ensemble lingo both *Sensors* and *Actuators* are *EventHandlers* which can behave periodically or sporadically; periodic events are those produced and consumed at a fixed rate (e.g. audio streams, video input and output) while sporadic events may be produced or consumed at any time, without any assumed regularity (e.g. motion requests or MIDI and text message exchange).

The former type is regulated by the *Environment* agent through an *EventServer*, which periodically notifies all registered *Actuators* of the beginning of a cycle (which works similarly to the DSP cycles in Pd or Jack, for instance). *Actuators* then notify associated *Reasonings* through the *needAction()* method, which establishes a deadline for data to be written in the corresponding *Memory* for the *Actuator* to produce in the next cycle. *Sensors* receive data from the corresponding *EventServer*, write the received data in the corresponding *Memory* and notify associated *Reasonings* through the *newSense()* method.

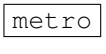
Sporadic events are treated straightforwardly, through direct communication between the related components: a *Reasoning* writes an *Event* in an *Actuator*'s *Memory*, then triggers the action that sends data to the *EventServer*, which in turn delivers the *Event* to the appropriate *Sensors*, who register it in their corresponding *Memories* and notify the appropriate *Reasonings* of the incoming sense data.

Specialized components have been made available for several common tasks, including *AudioSensors* / *AudioActuators* / *AudioMemories* and *MovementSensors* / *MovementActuators* / *MovementEventMemories*, that are treated by the corresponding *EventServers* which implement many alternatives for sound propagation and motion simulation within the virtual world, including realistic 3D spherical sound propagation (with the implied Doppler effect) and rotation and acceleration instructions considering friction, obstacles, etc. The use of these components in a Java-written *Reasoning* is straightforward, and the proposed interface between Ensemble and Pure Data makes these components accessible through Pd Reasoning patches.

3. LIBPD AND ENSEMBLE

Libpd comprises a series of C functions that wrap a subset of the Pd C API [17]. User interface, timing, threading and audio API are left out of the wrapper, allowing application code to use Pd as an embeddable Digital Signal Processing library. The libpd C code compiles to a dynamic library that has bindings for several languages such as Python, Java, Processing, and Objective-C, also supporting Android and iOS applications. The developer is then able to write applications that communicate with Pd patches, sending and receiving data and using patches as audio synthesis and prototyping systems [18, 19].

As opposed to what happens in traditional instances of Pd, patches loaded by libpd receive and send audio samples via arrays, sent as arguments to a processing method called by the user application. Also, internal clocks are incremented only when such method is called, meaning that audio sample calculations and DSP cycles are not affected by what is done in the application until the process method is called again. Because of this, objects that need time tracking to function properly, such as



will update their timers only when the application is processing Pd cycles. For example, if Pd was initialized with the default values of block size and sample rate, each call to the *process* method will advance Pd's internal clock by 64/44100 s or approximately 1.45 ms.

After a call to the *process* method, the application has to poll Pd for symbols, messages, floats, bangs, arrays and MIDI events, carry on its own processing cycles, and send its own symbols, messages, floats, bangs, arrays and MIDI events to the patch.

The PdBase class offers the Java bindings for libpd, thread synchronization and type conversion between Java and C. With the methods implemented in this class, it is possible to open patches, poll Pd for its current state, and process Pd DSP ticks.

To receive data from Pd, the application has to register the Pd symbols that it will listen to via the *subscribe* method in PdBase. The PdDispatcher class implements one callback function for each data type that Pd is able to send, and when Pd is polled for messages via the *pollPdMessageQueue* method in PdBase, it calls the appropriate methods of PdDispatcher for each data type that was sent to subscribed symbols. Symbols can be subscribed and unsubscribed to at any moment.

Ensemble extends the PdDispatcher class, overriding the callback methods to return encapsulated Pd data, which is processed by the PdEventServer class. This Event Server also parses messages received from Pd and encapsulates data from Ensemble to the patch, providing another layer in the access of the PdBase class, making future changes to the protocol easier to implement.

4. THE PD-ENSEMBLE PROTOCOL

With the libpd Java bindings, it is possible to divert all requests to any given agent reasoning (in Java) to a call to libpd that causes a Pd patch to be executed, and to collect any data produced by this patch back to the reasoning for further processing by Ensemble (through its actuators, event servers, etc). In such a context the Java-written reasoning is nothing more than a generic *wrapper* that requires only a few parameter inputs (e.g. the Pd patch filename that contains the actual reasoning).

In order for this wrapper-reasoning to work, the Pd patch has to adhere to a certain protocol, using specific Pd objects and a well-defined syntax to gain access to the structures contained in the agent's representation in Ensemble. This protocol should define Pd objects for accessing and modifying memories corresponding to existing sensors and actuators and for retrieving and storing data in a knowledge base, which are essential operations in the design of any reasoning. Furthermore, this protocol should also specify Pd objects to create and modify agents and components, and also to define characteristics of the virtual world that were formerly configured in an XML initialization file.

This protocol, defined in the sequel, should aim at complete transparency (from the user point-of-view) with respect to inner workings of the Ensemble framework, that will still be carried out by its Java kernel. In practical terms, no knowledge of the Ensemble Java classes and code should be required from the user of Ensemble through this Pd Ensemble protocol. Ideally, the Pd Ensemble user should not be even aware that any data is coming from or going to the Ensemble Java kernel outside Pd.

4.1 Environment

Ensemble applications can be configured and initialized by an XML file parsed by a loader class, where the user defines the application's components, such as Event Servers, Worlds and its Laws, Musical Agents and their Sensors and Actuators. The Ensemble-Pd interface allows these configuration parameters to be defined by message arguments sent to symbols that Ensemble subscribe to. These messages should be sent at the moment the patch is loaded by Ensemble's Pd instance.

Messages sent to the **global** symbol define Ensemble's clock, schedule and process modes; messages sent to the **environment** symbol define the Environment Agent, World and Laws; and messages sent to the **add_agent** symbol define Musical Agent names that will be added to the application, and whose definition should be in subpatches inside the configuration patch. Table 1 shows the Pd symbols and objects used in the initialization, their arguments and usage.

Symbol/Object	Arguments	Usage
global	<clock_mode> <process_mode> <scheduler_threads>	Sets execution parameters
environment	<class> <world> <*world_class*> [<*world_law*>]	Sets environment and world parameters
add_agent	<name>	Adds an Agent
[r *<agent>*/start]		Receives agent startup bang

Table 1. Application Startup Symbols.

4.2 Sensors and Actuators

Every Musical Agent added to the Pd application has its own subpatch, where its Reasoning, Actuators, Sensors, Memories and Knowledge Base are defined and configured via messages, which are sent to Ensemble symbols when the framework is ready to start processing that Agent. This is done to prevent Ensemble from losing the agent configuration messages, because at the moment of loading the patch Ensemble has not yet received the agents' names, and so could not have subscribed to their symbols. Table 2 shows the Pd symbols and objects used in Sensor and Actuator creation and communication, their arguments and usage.

Ensemble starts agent processing by sending (through libpd) a bang to the symbol `<agent>/start` (see Table 1) which will be received in the application's patch by a Pd object

Symbol/Object	Arguments	Usage
add_actuator	<name> <type> [<scope>]	Adds a new Actuator
add_sensor	<name> <type> [<scope>]	Adds a new Sensor
remove_actuator	<name>	Removes an Actuator
remove_sensor	<name>	Removes a Sensor
[s *actuator_name*]	<data> [<scope>]	Writes data to *actuator_name*
[r *sensor_name*]		Receives data from *sensor_name*

Table 2. Actuators and Sensors Symbols.

`r <agent>/start`

Messages in the patch connected to this object should contain the names and parameters for the desired components and Knowledge Base facts. Messages in the patch sent to the symbol corresponding to the agent name are used to create components; for instance sending a message to a symbol `<agent>` with contents **add_actuator** or **add_sensor** initialize (in Ensemble) the corresponding Sensors and Actuators, and also define the type of Event that this component will handle; optionally this message can define a scope, allowing for private communication between selected actuators and sensors.

4.3 Knowledge Base

Adding facts to the Knowledge Base at agent startup time is done by sending a message with the fact name and value to the symbol **new_fact**. Facts can later be recovered by sending messages to the object

`read_fact <agent_name>`

which receives a fact name in its inlet and returns the value in its outlet (assuming the fact exists in the agent's Knowledge Base). Updating the fact in the agent's Knowledge Base is done via messages to the object

`update_fact <agent_name>`

that receives a fact name and new value in its inlet and updates the Knowledge Base accordingly. In Table 3 are the Pd symbols and objects used in creating, accessing, removing and updating a fact in the Agent's Knowledge Base, their arguments and usage.

4.4 Memories

In the same way, different types of Agent Memories can be created and read from with messages to the symbols **new_memory** and **read_memory**. To read from a memory, a message must specify a memory name and a time value; when writing to a memory (e.g. an actuator's memory), the value sent to the symbol **write_memory** is written in the specified Memory at the current processing time.

Symbol/Object	Arguments	Usage
new_fact	<agent> <name> <value>	Adds a new Fact
update_fact	<agent> <name> <value>	Updates a fact if it exists
remove_fact	name	Removes a Fact
[read_fact]	<agent> <name> <value>	Reads a Fact and returns a value

Table 3. Knowledge Base Symbols.

Table 4 presents the Pd symbols and objects used in Memory creation, writing and reading, their arguments and usage.

Symbol/Object	Arguments	Usage
new_memory	<agent> <name> <type>	Adds a new Memory
write_memory	<agent> <name> <value>	Writes to a Memory at current time
read_memory	<agent> <name> <time>	Reads a Memory and returns a value

Table 4. Memory Symbols.

5. PD-ENSEMBLE EXAMPLES

A minimal Pd-Ensemble application should start by defining the internal clock and processing modes of Ensemble, the environment agent and world, and how many agents of a given class the application will instantiate. In a patch, these parameters are defined by sending messages to the symbols shown in Table 1. The patch in Figure 1 shows these messages loaded at startup time and a subpatch corresponding to an agent.

Figure 1. Pd-Ensemble initialization patch

The agent reasoning is defined inside the subpatch, in this case **agent1**, which will also define the agent components. Figure 2 shows the **agent1** patch, that adds one sensor and one actuator of the type *audio*.

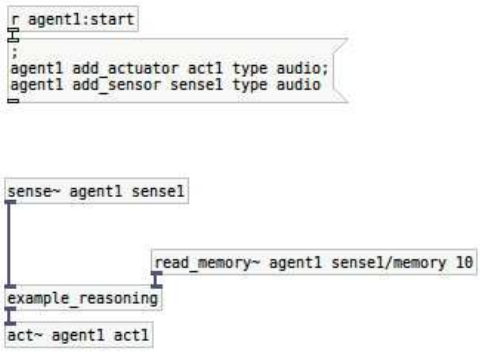


Figure 2. Agent configuration and reasoning

The reasoning defined by this patch is also shown in Figure 2, and uses the abstraction in Figure 3 to process two audio signals using an FM structure. These signals are received from an audio sensor (immediate input), and from reading an audio memory corresponding to the same sensor with a 10-second delay, using the Pd objects

`sense~`

`act~`

`read_memory~`

shown in the patch. The Pd abstractions for these objects are shown in Figures 4, 5 and 6, and serve as wrappers of the Pd-Ensemble Protocol.

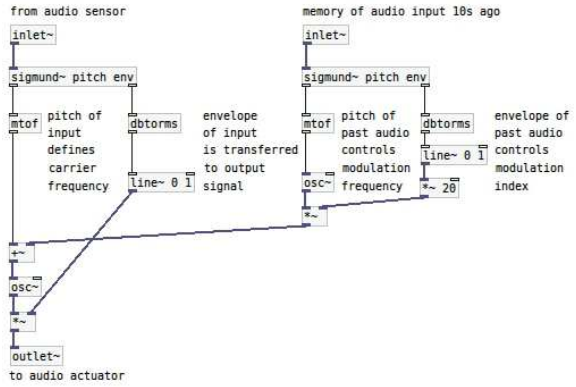


Figure 3. Signal processing Pd abstraction used in this example

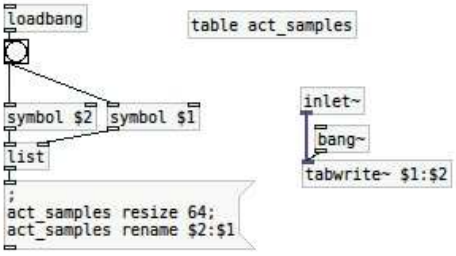


Figure 4. Pd abstraction for writing a signal to an actuator

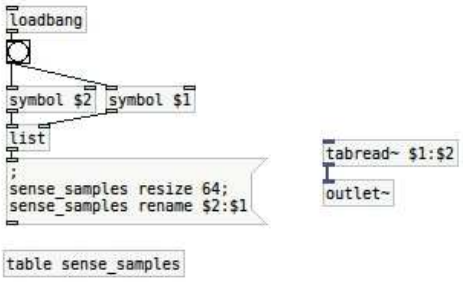


Figure 5. Pd abstraction for receiving a signal from an audio sensor

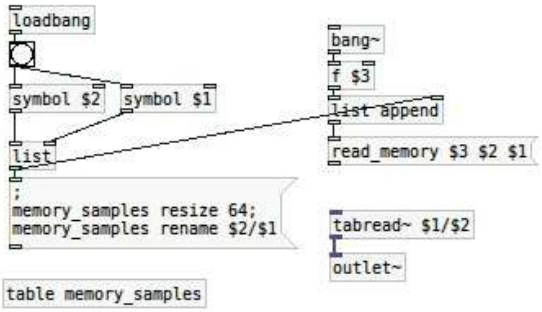


Figure 6. Pd abstraction for reading from memories of an agent

For a given Ensemble configuration Pd patch, one can run Ensemble with that patch by passing the patch name to the Loader class, as in `ensemble.tools.Loader -patch patch_name`, or in a full command-line example:

```
$ java -cp ../lib/libpd.jar:  \
../lib/ensemble_apps.jar:  \
../lib/ensemble.jar:  \
../lib/NetUtil.jar:  \
../lib/jade.jar \
ensemble.tools.Loader -patch patch_name
```

6. DISCUSSION AND CONCLUSIONS

The protocol here defined enables the creation of multiagent musical applications in Ensemble by users with knowledge of Pure Data programming. Objects and messages in Pd have been presented which allows a Pd patch to define several characteristics of the virtual world, including topological and physical aspects, to create agents and components, such as sensors and actuators, and to manipulate such components, allowing the behaviour of agents to be entirely defined via Pd patches.

The Ensemble Pd interface does not contemplate several higher-level structures that affect the virtual environment as a whole. For instance, with Java an Ensemble user is able to define new types of data for agent communication that are not currently supported, along with the physical laws that govern the distribution of such data among regular agents and the environment agent. The user can also create (in Java) alternatives to existing laws, such as arbi-

trary sound propagation rules limited only by one's imagination; for instance, sounds that accelerate away from each sound source or boomerang-like sounds that bounce back toward the source a while after being produced are easily defined in Java, but are still off-limits to a pure Pd-Ensemble user.

These possibilities should become available to a Pd user in one of two possible forms, which are the theme of future work. Either the protocol here defined is enlarged in order to accommodate all sorts of substitutions in the Ensemble kernel (e.g. virtual world laws, environment policies) for surrogate Pd patches, or the entire Ensemble kernel is rewritten in C as a series of Pd externals.

Acknowledgments

Authors would like to thank the support of the Sonology Research Nucleus (NuSom) of the University of São Paulo, and PIBIC/CNPq project # 2013/1046.

7. REFERENCES

[1] P. M. Todd and G. M. Werner, "Frankensteinian methods for evolutionary music composition," in *Musical Networks: Parallel Distributed Perception and Performance*, 1999, pp. 313–340.

[2] M. Spicer, "AALIVENET: an agent based distributed interactive composition environment," in *International Computer Music Conference*, 2004, pp. 1–6.

[3] L. K. Ueda and F. Kon, "Andante: Composition and performance with mobile musical agents," in *Proceedings of the International Computer Music Conference*, 2004, pp. 604–611.

[4] D. Murray-Rust, A. Smaill, and M. Edwards, "MAMA: An architecture for interactive musical agents," in *Proceeding of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29–September 1, 2006, Riva del Garda, Italy*. IOS Press, 2006, pp. 36–40.

[5] D. Bisig, M. Neukom, and J. Flury, "Interactive Swarm Orchestra-A Generic Programming Environment for Swarm Based Computer Music," in *Proceedings of the International Computer Music Conference. Belfast, Ireland*, 2008.

[6] L. F. Thomaz and M. Queiroz, "A framework for musical multiagent systems," in *Proceedings of the Sound and Music Computing Conference*, Porto, Portugal, 2009, pp. 213–218.

[7] I. Whalley, "Software agents in music and sound art research / creative work: Current state and a possible direction," *Organized Sound*, vol. 14, no. 2, pp. 156–167, 2009.

[8] L. F. Thomaz and M. Queiroz, "Ensemble: Implementing a musical multiagent system framework," in *Proceedings of the Sound and Music Computing Conference*, Padova, Italy, 2011, pp. 198–205.

[9] G. L. Ramalho, P. Y. Rolland, and J. G. Ganascia, "An artificially intelligent jazz performer," *Journal of New Music Research*, vol. 28, no. 2, pp. 105–129, 1999.

[10] S. Dixon, "A lightweight multi-agent musical beat tracking system," in *Proceedings of the Pacific Rim International Conference on Artificial Intelligence*, 2000, pp. 778–788.

[11] J. McCormack, "Eden: An evolutionary sonic ecosystem," *Advances in Artificial Life*, pp. 133–142, 2001.

[12] P. Dahlstedt and M. Nordahl, "Living melodies: Coevolution of sonic communication," *Leonardo*, vol. 34, no. 3, pp. 243–248, 2001.

[13] E. R. Miranda, "Emergent sound repertoires in virtual societies," *Computer Music Journal*, vol. 26, no. 2, pp. 77–90, 2002.

[14] M. Gimenes, E. Miranda, and C. Johnson, "The development of musical styles in a society of software agents," in *Proceedings of the International Conference on Music Perception and Cognition*, 2006.

[15] L. L. Costalonga, R. M. Vicari, and E. M. Miletto, "Agent-based guitar performance simulation," *Journal of the Brazilian Computer Society*, vol. 14, pp. 19–29, 2008.

[16] P. A. Sampaio, G. Ramalho, and P. Tedesco, "Cinbalada: a multiagent rhythm factory," *Journal of the Brazilian Computer Society*, vol. 14, pp. 31–49, 2008.

[17] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner, "Embedding pure data with libpd," in *Proc Pure Data Convention 2011*, 2011.

[18] K. Jolly, "Usage of pd in spore and darkspore," in *Proc Pure Data Convention 2011*, 2011.

[19] S. Jorda, M. Kaltenbrunner, G. Geiger, and R. Bencina, "The reactable*," in *Proceedings of the international computer music conference (ICMC 2005), Barcelona, Spain*, 2005, pp. 579–582.