MEPHISTO: A Source to Source Transpiler from Pure Data to Faust

Abdullah Onur Demir and Hüseyin Hacıhabiboğlu
Graduate School of Informatics
Middle East Technical University (METU)
Çankaya, Ankara, Turkey, TR-06800
aaonurdemir@gmail.com, hhuseyin@metu.edu.tr

ABSTRACT

This paper introduces Mephisto, a transpiler for converting sound patches designed using the graphical computer music environment Pure Data to the functional DSP programming language Faust. Faust itself compiles into highly-optimized C++ code. The aim of the proposed transpiler is to enable creating highly optimized C++ code embeddable in games or other interactive media for sound designers, musicians and sound engineers using PureData in their work flows and to reduce the prototype-to-product delay. Mephisto's internal structure, conventions, limitations and performance are presented and discussed.

1. INTRODUCTION

Sound synthesis is crucially important not only for electro-acoustic music but also for games and virtual reality applications. High quality audio has a decisive role while creating realistic environments and evoking interest in user experience in video games [1]. There are two common techniques used to create high quality sounds in games. As a first technique, prerecorded clips, also known as Foley sounds, are used. They are the same as sampling and can be modified heavily by processing their samples. Although prerecorded clips provide perfect realism and low computational cost, memory footprint becomes the main bottleneck. Prerecorded clips should reside in memory since the I/O latency of the disk is unacceptable. Besides, loading every sound in physical memory is not a very good solution because of the fact that typically a limited amount of hardware resources is allocated for sounds in games. Moreover, it is particularly hard if not impossible to record sounds such as jet engines, gunshots, rain or wind due to physical and practical reasons.

The second alternative is based on the concept of parametric sound synthesis with which impressive results can be obtained by algorithmic means. Such algorithms allow the model-based generation of hard-to-record sounds by simple signal generators and signal processing methods yet provide convincing results. This approach extends the sound designer's palette by providing her with the means to construct and control virtual sound objects. Synthesized sounds are computer programs that can be executed and parametrically adjusted in real time [2].

1.1 Pure Data

Pure Data (Pd) [3] is a popular *data flow* programming language with which composers, performers and developers can synthesize sounds without writing code but by using graphical objects and connections between them. Pd does real time computations using a Max-like message interpreter and scheduler and operates on vector samples in order to minimize the interpretation overhead and to satisfy the needs of the real time audio applications. However, sample level computations have to be carried out by using external plug-ins or primitives. Hence, Pd programs depend on a run-time environment. As a result, although not impossible, embedding interpreted Pd programs in games or other systems is generally difficult and inefficient in comparison with code written directly in compiled languages like C or C++ [4].

1.2 Faust

Faust (Functional AUdio STreams) is a functional, *block diagram* programming language designed and used specifically for processing digital signals in real-time [5]. It can be used to create high-performance audio applications and plug-ins. While Faust is well-suited for signal processing, it lacks sufficiently elaborate control mechanisms offering only basic user interface elements like buttons, sliders and number boxes. Faust is designed 1) to be highly expressive, 2) to have clean mathematical semantics, and 3) to be highly efficient [6, 7].

1.3 Motivation

Data flow languages such as Pure Data or Max are popular since programming audio graphically is much easier than writing code. Besides, such languages allow changing parameters and observing the results on-the-fly. However, a particular problem with Pd is that the developed algorithms are difficult to integrate in the final product due to the reduced performance [1], necessity to bundle the run-time environment with the final product (e.g. a game or a mobile app) and incompatibility with existing development frameworks.

[1] PureData is reported to be roughly three orders of magnitude slower than its C equivalent for multiplying floating point numbers [8].

In contrast, programs written in Faust can be directly translated to optimized C++ code, can be embedded into a final product in a more straightforward way and also provide a higher performance. However, the main drawback of the Faust language is that the programmer has to learn functional programming concepts, syntax and semantics of Faust and mathematical descriptions of signals to be able to code in Faust.

This paper presents Mephisto, a source-to-source transpiler from Pure Data to Faust which aims to bridge the gap between the two programming languages and to facilitate easier design and integration of audio algorithms in relevant software development processes.

The paper is organized as follows. Section 2 presents the works which bring solutions to the problems in a similar way with Mephisto. Section 3 talks about how Mephisto is designed and how it can be used to generate Faust code from Pure Data patches. Section 4 talks about the conventions that Mephisto uses along with its limitations. The performance of Mephisto/Faust generated code in comparison with Pd itself and libpd is discussed in Section 5. Section 6 talks about possible directions for Mephisto and Section 7 concludes the paper.

2. RELATED WORK

Two other similar tools to what is presented in this paper exist. These are PUre DAta Compiler (*PuDaC*) and *libpd*. A short overview of these tools are given in this section.

2.1 PuDaC

PuDaC (PUre DAta Compiler) is a compiler created in order to address the performance issues pertaining to Pd [8]. PuDaC considers data as if it consists of two parts: High-bandwidth (audio) and low-bandwidth (control) signals. The equivalent of each Pd object in the patch is transformed into C language. The connections are translated as function calls. As a result, the Pd patch is transformed into a C program which can run efficiently on embedded systems with limited computational resources. However, the resultant C program is not optimized specifically for audio processing in contrast to C++ code that can be generated by Faust and is thus suboptimal in terms of performance.

2.2 libpd

libpd [9] is a free and open source software library which enables the usage of *PureData* almost everywhere from embedded devices to phones and computers.

libpd is not a substitute of Pd but it is the embedded version of Pd itself. Since it is embeddable, it can run on any hardware that can run native code. Hence, Pd patches can be incorporated within games, Android or iOS applications, or programs written in C running on embedded Linux systems including microcontroller boards like Intel Galileo. *libpd* helps Pd run in the musical application or the game as a compiled program.

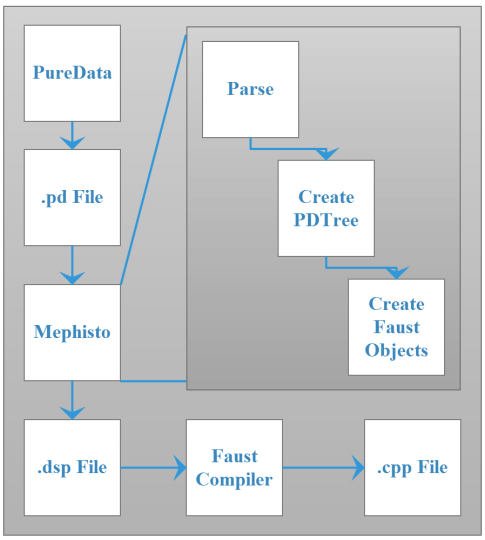

Figure 1: Transpiling pipeline of Mephisto

3. MEPHISTO: A SOURCE TO SOURCE TRANSPILER

Mephisto is developed in order to enable Pure Data users to incorporate the audio algorithms they design into games and other applications by utilizing the highly optimized C++ code created by Faust. With this motivation, Mephisto transpiles PureData sources into Faust sources and the creation of optimized C++ code is then left to the Faust compiler.

Transpilation process consists of four steps: 1) parsing the Pd source, 2) creating Pd object tree and traversing it, 3) creating Faust source on-the-fly while traversing the tree, and 4) compiling the transpiler-generated Faust code to obtain optimized C++ code. A visual representation is shown in Fig. 1.

3.1 Parsing

ANTLR v4 is a powerful parser generator used to read, process, execute, or translate structured text such as program source code, data, and configuration files [10]. ANTLR v4 also has the capability to carry out lexical analysis, token generation, and parse tree creation. Given these advantages that simplify and integrate the work flow, we implemented the Pure Data parser with ANTLR v4.

Each object positioned on the canvas of a Pd patch is represented as a single row in the source file. Each row contains the definition of the corresponding object and its parameters. By using the format specification of Pd and ANTLR v4, a formal language description, i.e. the *grammar* of Pure Data, was created. This grammar is used by ANTLR for generating a standalone Java program that processes Pd source files and builds a data structure representing the input (i.e. *parse tree*). The row nodes of an example parse tree are shown in Fig. 2. Additionally, ANTLR v4 automatically generates *tree walkers* (listeners) that can be used to visit the nodes of the parse tree to create a Pd object tree. A parse tree listener interface consists of simple event listeners triggered by the built-

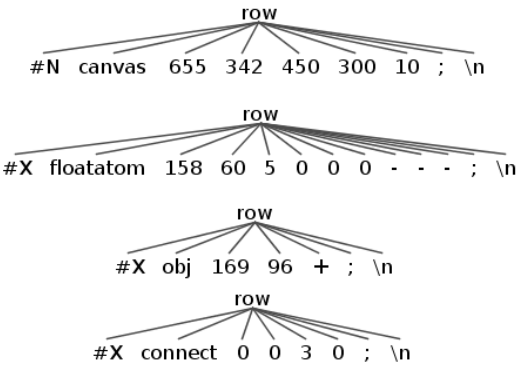

Figure 2: Each row represents a Pd object. Each element in a row represents arguments of that object. All tokens are generated from Pd source in Mephisto's parsing stage

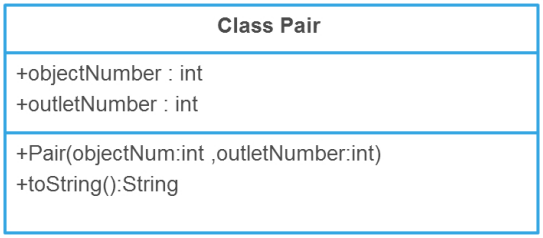

Figure 3: Pair Class representing connections in a Pd patch

in tree walker [10]. ANTLR v4 generates enter and exit methods for each node in the parse tree. Enter event is triggered when tree walker enters a node. Exit event is triggered when it completes traversing the node's children nodes and leaves it. By using these listeners, current node is determined and can be processed based on its context. Details of the file format specification of Pure Data source used by Mephisto can be found in Pd's distribution site [11].

3.2 Creation and the Traversal of Pd Object Tree

In order to define the semantics of Pd objects and the connections between them, we created a data structure to form a tree which will henceforth be called a *PDTree*. The data structure is shown in Figs. 3 and 4

`Pair` class represents an outgoing connection of a Pd object. `objectNumber` represents unique ID of it and `outletNumber` represents which outlet of it belongs to the referred connection.

`PDObject` class represents a Pd object itself. `defaultVal` keeps the default value of the Pd object as statically set in the patch (e.g. the initial frequency of an oscillator). For multiple initialisation arguments, `args` array is used instead of the `defaultVal` attribute. The most important part of this class is the `objectInlets` attribute which is a hash map whose keys are representing its inlets. Each key holds a `Pair` instance. For example, index `0` holding a `Pair` object with the values `objectNumber=2` and

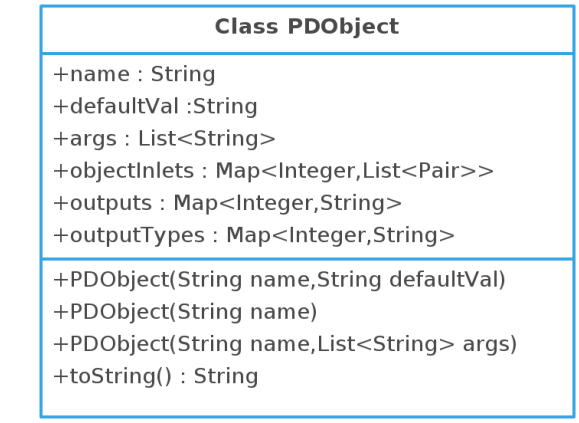

Figure 4: PDObject Class representing Pd objects in a Pd patch

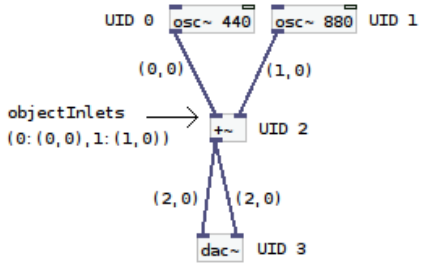

Figure 5: Visualization of usage of Pair Class and PDObject Class instances

`outletNumber=0` means that the first outlet of another Pd object whose `UID` is `2` is connected to the first inlet of the present Pd object. The `outputs` attribute represents the outlets of the Pd object. Each key refers to each outlet of the Pd object mapping the key `i` to outlet `j` and so on. Fig. 5 illustrates the data structure.

While a tree walker walks on the ANTLR v4 generated parse tree, it dispatches listener events. We use listeners automatically generated by ANTLR which are `enterRow` and `exitFile`. `PDObject` instances are created in `enterRow` callbacks which are called with a context argument keeping the children nodes of the row node being processed in the parse tree. Since all arguments and parameters which are children of that node are found in the context, a `PDObject` with a unique ID can easily be created without explicitly traversing its children. After the creation of a `PDObject` instance, it is pushed to a global list in order to be accessed later. Additionally, if `enterRow` callback is called for a connection token, a `Pair` instance is created representing a connection between Pd objects and is inserted within the corresponding `PDObject`'s hash-map.

At the end of the traversal of the parse tree, a `PDTree` is created which has `dac~` object in the Pd patch as its

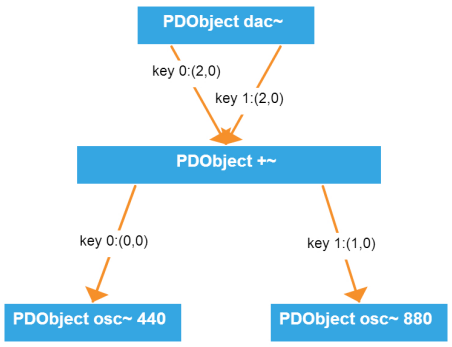

Figure 6: PDTree. The numbers represent traversal order

root. At the end of the tree traversal process, `exitFile` callback function is called and a second traversal is started beginning from the root of the `PDTree` following depth-first search (see Fig. 6).

3.3 Traversal of `PDTree` and creation of Faust Code

The second traversal is started on the `dac~` object by the call of the `exitFile` callback function. At first, the hashmap `objectInlets` of root object, `dac~`, is scanned. The first key is always `0` representing the first inlet of the object. Additionally, the value of the key represents a `Pair` object indicating a connection incident from an outlet of another `PDObject` instance. The traversal is carried out by recursion by a function having the definition, `createObject_setOutput(int objectNumber,int outletNumber)`.
After giving the value(Pair) of the key `0` to this function as `<Pair instance>.objectNumber` and `<Pair instance>.outletNumber` it sets the outputs and `outputTypes` of that object and finally returns the Faust equivalent outlet value of the processed `PDObject` instance. This procedure is applied to each key stored in the `objectInlets` attribute for each `PDObject` instance.

Since `dac~` object in Pure Data represents the sound hardware that will output signals incident to its inlets, it is transpiled to a Faust `process` which forms the entry point of the generated Faust program. The inlets of the `dac~` object are mapped to left and right channels in the generated Faust program. Transpilation process is completed when the tree traversal ends.

3.3.1 createObject_setOutput(int objectNumber,int outletNumber) Function

This function is at the core of Mephisto and is used to create the Faust equivalents of Pd objects. The first argument (`objectNumber`) defines the `UID` of the `PDObject` instance stored in the aforementioned global list. The second argument (`outletNumber`) defines the outlet value to be returned. The function first reads the instance from the list by its `objectNumber`. After this is obtained and the name of the object is checked, the function calculates values incident from other connected objects into its inlets by using the same function recursively. After obtaining all incoming values for each inlet, a Faust function definition is created. Since Faust is a functional programming language, the defined function's name, complete with its arguments, is returned as an output.

3.4 Compiling transpiler-generated Faust code

On a system where Faust is installed, the resulting `.dsp` file can be compiled and a `.cpp` file can be generated. As a result, a well optimized C++ code can be created without writing any lines of code using Mephisto. The generated C++ code can be embedded in a rather straightforward way in any desired software project written using the C++ language.

4. CONVENTIONS AND LIMITATIONS

There are significant differences between Pure Data and Faust. Mephisto uses certain conventions to reconcile these differences. At its present version, Mephisto also has certain limitations. These conventions and limitations are discussed below:

1. Mephisto ignores the "cold inlet" mechanism of Pure Data. Since Faust is a language tailored primarily for processing audio streams, everything is considered as signals and there is no messaging or control mechanism except for elementary user interfaces. Hence, the transpiling process ignores the cold inlet semantic of Pd while translating it into Faust code.

2. Mephisto will not transpile Pd patches unless the patch includes a `dac~` object. Therefore, Pd patch to be transpiled should be formed as a connected tree [12] in which there is a `dac~` object. Since Mephisto transpiles Pd patches by constructing a tree, objects which are not included in this connected tree will be ignored by Mephisto.

3. Mephisto's main aim is to transpile Pd patches focused on signal blocks and control blocks are not covered. However, simple control mechanisms like number box, message, trigger, pack and unpack are provided. In addition, mathematical and logical operators in both the control and the signal blocks in Pd are also implemented in Mephisto. This is done by converting these to signal blocks since both messages and signals in Pd have to be represented as signals in Faust. Hence, all control logic, except for elementary controls, should be implemented separately in C++ (or any other programming language that Faust would support in the future) to incorporate within the code in which the generated C++ code will be embedded. Consider the patch in Fig. 7, for example. The patch synthesizes the standard CCITT dialing tone [13] which is the sound heard just after picking up the phone. It consists of nearly all signal objects except for two, which are messages that control starting and stopping the

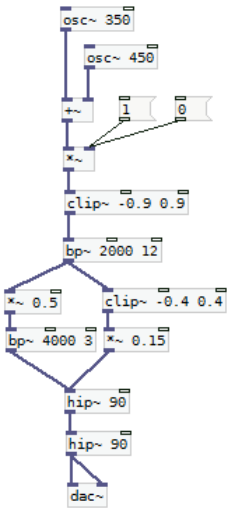

Figure 7: Dialing tone patch synthesizing the standard CCITT dialing tone

generated output. These two message objects can be transpiled into check-boxes in Faust code in order to preserve content integrity. However, these control statements should be properly implemented and connected to a (possibly event-driven) control code in which they will be embedded. Faust code automatically generated by Mephisto for the CCITT patch (Fig. 7) by Mephisto as well as the block diagram representation of the same Faust code are given in the Appendix.

4. Mephisto does not yet support sub-patches and external objects as it is designed to parse only one Pd source. Allowable objects are predefined in the transpiler and others are not recognized. Therefore, Pd projects that include sub-patches or external objects should be flattened to a single Pd patch.

5. Fig. 8a shows the regular convention in Pd to generate a counter. This counter incorporates a `float` object which is not recognized by Mephisto. In addition, the structure of the counter violates the tree structure of `PDTree` since it has a cycle. Since trees are defined as graphs which do not include cycles and that cycles will cause Mephisto to enter infinite recursion, counters cannot be created as a direct translation. In order to alleviate this problem, Mephisto requires that a special Pd patch, `fcounter`, be used which is an abstraction of the patch shown in Fig. 8b.

5. PERFORMANCE

A performance comparison of three different ways to execute algorithms designed in Pure Data is given in this section. Specifically, three different conditions were tested: 1) Pure Data natively running a patch, 2) the same patch executed via the embeddable Pure Data library *libpd* [9] and 3) C++ code generated by Mephisto and

(a) Conventional counter used in most of the Pd patches

(b) Mephisto Fcounter abstraction sub-patch. Used instead of conventional counter

Figure 8: PDObject and Pair classes

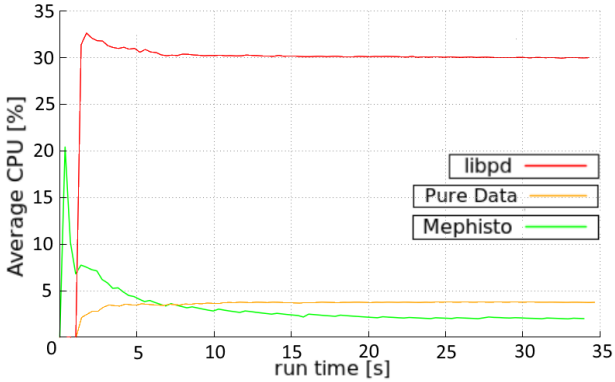

Figure 9: The average CPU utilization plots of the three tested conditions.

Faust, cross-compiled with JACK [14]. In each case the relevant process was isolated and the average CPU utilisation was obtained by the profiler, Audria [15]. The patch that was used in benchmarking these three cases consists of the product of the outputs of two oscillators for a duration of 30 s. The process was tracked for 35 s in each case. At the sampling rate of 44.1 kHz, this amounts to the generation of 2,646,000 floating point numbers and 1,323,000 floating point multiplications in total. Fig. 9 shows the average CPU utilization for each case. It may be observed that the highly optimized code generated by Mephisto and Faust outperforms both the native Pure Data and libpd-based implementations. In the case of Pure Data only, the average CPU utilization starts at zero and plateaus at slightly less than 4%. In the case of libpd, the same process results in around 30% utilization. For C++ code generated via Mephisto by Faust, the average utilization starts at 20% and quickly falls to just above 2%. The peak in the CPU utilisation curve for *Mephisto* between t=0 and t=1 is thought to be caused by external libraries (JACK and Qt) used in compiling the C++ code to create a working program. These results indicate that generating C++ code by Faust via Mephisto is promising in terms of the potential performance gains that it can provide.

6. FUTURE WORK

Mephisto is at a stage that basic sound synthesis algorithms designed in Pure Data can be transpiled into Faust code in order to obtain highly optimized C++ code for use in games and other applications. However, the limitations outlined in the previous section remain to be solved.

A possible solution to cold inlet and messaging problem

can be developed by designating an impulse signal as a bang message. In Faust, the lack of messages means that numbers are treated as signals and there is no gating mechanism as in Pd. The deficiency caused by Faust's lack of such control mechanisms could be filled by creating specialized Faust functions and revising the traversal mechanism.

It is critical that the Pd patch does not include any cyclic connections. However, Mephisto can be further developed so as to allow transpiling Pd patches containing disjoint or cyclic trees [16] with no constraints on the root objects.

In addition, Pd abstractions for Faust code could be written for Mephisto so that it could deduce what the semantic of the abstraction is and does not care about its internal structure. Faust implementation can then be created and added to the source code of Mephisto. The example for `fcounter` can act as a starting point.

The work presented in this paper acts as a proof-of-concept showing that transpilation from Pure Data to Faust is possible and thus the present version of Mephisto does not cover more than a few Pd objects. Any additional object that is added to Mephisto would increase the palette of objects available to the Pd programmers. For this reason, Mephisto will be released with the GPL v3.0 license and will be made open source in a GitHub repository [17] to allow other developers to work on it. Additionally, further examples including fire, jet engine and wind sound adapted from [2] reside in the repository.

7. CONCLUSION

Pure Data is both a Turing complete programming language and a very useful tool for designing audio algorithms from scratch. However, since Pure Data is an interpreted (as opposed to compiled) language, it presents important performance issues. Therefore, once the prototype algorithms are designed, the designer or the developer still has to carry the burden of writing code to integrate the algorithm in a final standalone app such as a game or an audio app typically using another high-level language such as C++. While this approach is feasible, the effort required from the C++ programmer is considerable and a solution that can automatically generate C++ code from Pure Data code has practical benefits. A transpiler from Pure Data to Faust, named Mephisto, is described in this paper that can achieve this. While Mephisto can generate Faust code from Pure Data patches, Faust is used as an intermediate language which already has a compiler that can generate C++ code.

8. ACKNOWLEDGMENTS

This work was supported by the Middle East Technical University Faculty Research Grant, BAP-08-11-2013-057.

9. REFERENCES

[1] D. B. Lloyd, N. Raghuvanshi, and N. K. Govindaraju, "Sound synthesis for impact sounds in video games," in *Proceedings of the Symposium on Interactive 3D Graphics and Games 2011*. ACM, 2011.

[2] A. Farnell, *Designing Sound*. Cambridge, MA, USA: MIT Press, 2010.

[3] M. Puckette, "Pure Data: Another integrated computer music environment," in *Proc. Second Intercollege Comp. Mus. Concerts*, Tachikawa, Japan, 1996, pp. 37–41.

[4] Y. Orlarey, D. Fober, and S. Letz, "FAUST: an efficient functional approach to DSP programming," in *New Computational Paradigms for Computer Music*. Editions Delatour, Paris, France, 2009.

[5] ——, "An algebra for block diagram languages," in *International Computer Music Conference*, Göteborg, Sweden, September 2002.

[6] J. O. Smith III. (2010) Audio signal processing in Faust. [Online]. Available: https://ccrma.stanford.edu/jos/aspf

[7] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[8] R. N. Jacobs, "A wireless sensor-based mobile music environment compiled from a graphical language," Ph.D. dissertation, MIT Media Lab, Cambridge, MA, USA, September 2007.

[9] libpd ≫ about. [Online]. Available: http://libpd.cc/about/

[10] T. Parr, *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2012.

[11] (2004, October) Unofficial PD v0.37 fileformat specification. [Online]. Available: http://puredata.info/docs/developer/PdFileFormat

[12] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*. Macmillan London, 1976.

[13] International Telecommunications Union (ITU), *Application of tones and recorded announcements in telephone services*, CCITT Recommendation E.182, 1998.

[14] JACK Audio Connection Kit. [Online]. Available: http://jackaudio.org/

[15] audria - A Utility for Detailed Resource Inspection of Applications. [Online]. Available: https://github.com/scaidermern/audria

[16] M. Nilsson and H. Tanaka, "Cyclic tree traversal," in *Third International Conference on Logic Programming*. Springer, 1986, pp. 593–599.

[17] A. O. Demir. (2015) Mephisto Source Code. [Online]. Available: https://github.com/aonurdemir/Mephisto

APPENDIX

Mephisto-generated Faust Code for the CCITT dialling tone patch

CCITT dialling tone patch was given as an example earlier in the paper in Fig. 7. Faust code automatically generated by Mephisto for this patch is:

```
CCITT.dsp
import("music.lib");
import("math.lib");
import("filter.lib");

osc1=osc(350);
osc2=osc(450);
checkbox3=checkbox("1");
msg3 = checkbox3 * 1;
checkbox4=checkbox("0");
msg4 = checkbox4 * 0;
clip7(s) = if (s < (-0.9), (-0.9) , if ( s>0.9 ,0.9 ,s) );
resonbp8=clip7(((((osc1+osc2))*((msg3)+(msg4))))):resonbp(2000,12,1);
clip10(s) = if (s < (-0.4), (-0.4) , if ( s>0.4 ,0.4 ,s) );
resonbp11=((resonbp8)*0.5):resonbp(4000,3,1);
resonhp13=(resonbp11+((clip10((resonbp8)))*0.15)):highpass(1,90);
resonhp14=(resonhp13):highpass(1,90);
process=resonhp14,resonhp14;
```

Figures below show the different block diagrams generated for the same Pure Data patch. The figures were generated using FaustWorks IDE. The figures follow the tree traversal order as explained in the text, starting with the `dac~` object and following through to the `osc~` objects and the message boxes (emulated using checkboxes in Mephisto-generated Faust code.).

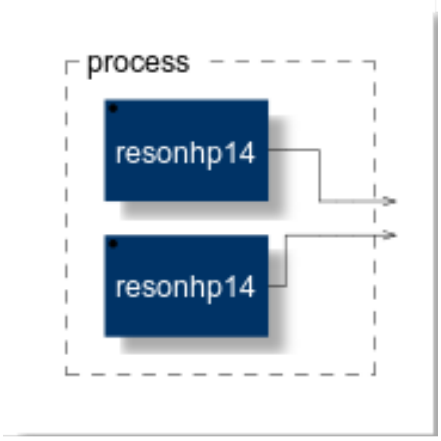

Figure 10: Equivalent of ”dac∼” object

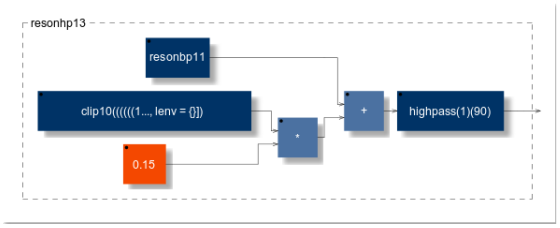

Figure 12: Equivalent of ”hip∼ 90” object

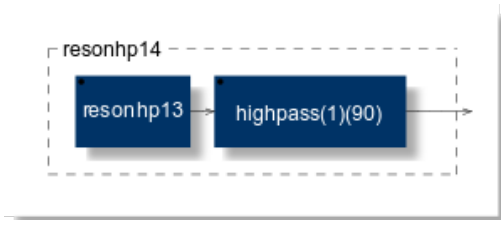

Figure 11: Equivalent of ”hip∼ 90” object

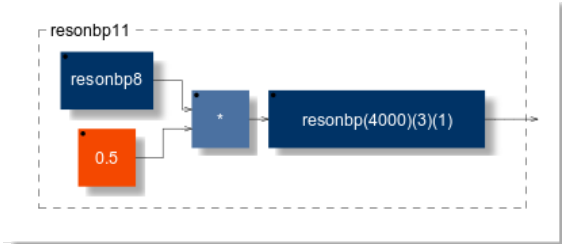

Figure 13: Equivalent of ”bp∼ 400 3” object

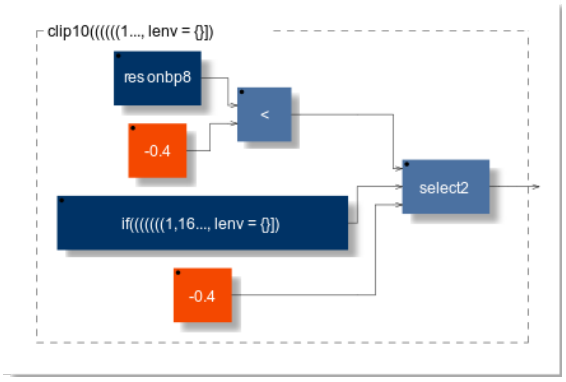

Figure 14: Equivalent of ”clip∼ -0.4 0.4” object

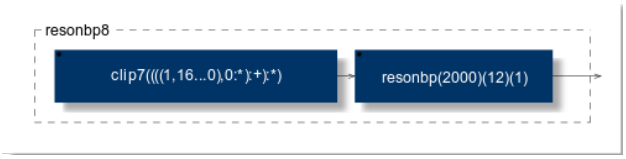

Figure 15: Equivalent of ”bp∼ 2000 12” object

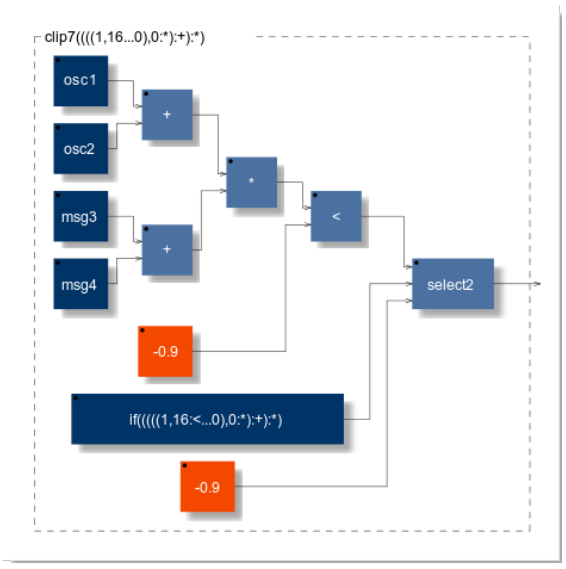

Figure 16: Equivalent of ”clip∼ -0.9 0.9” object

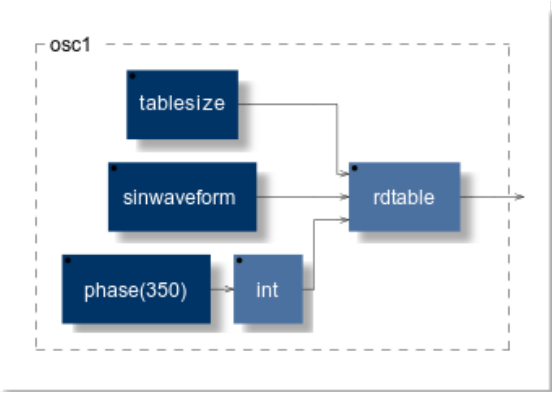

Figure 17: Equivalent of ”osc∼ 350” object

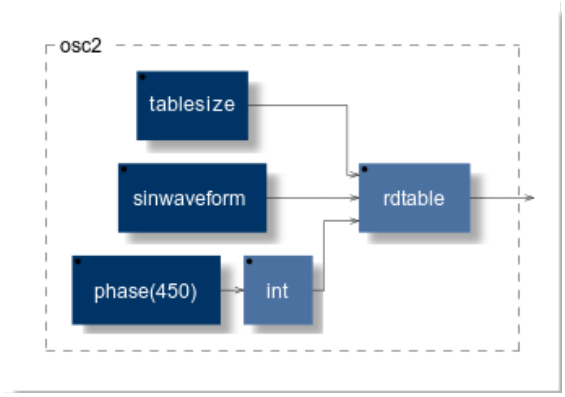

Figure 18: Equivalent of ”osc∼ 450” object

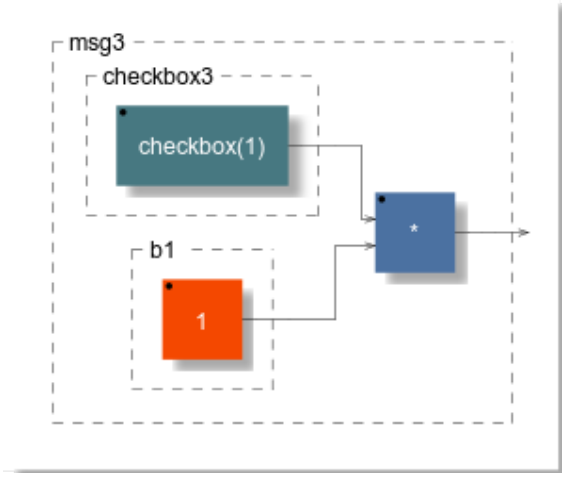

Figure 19: Equivalent of message object ”1”

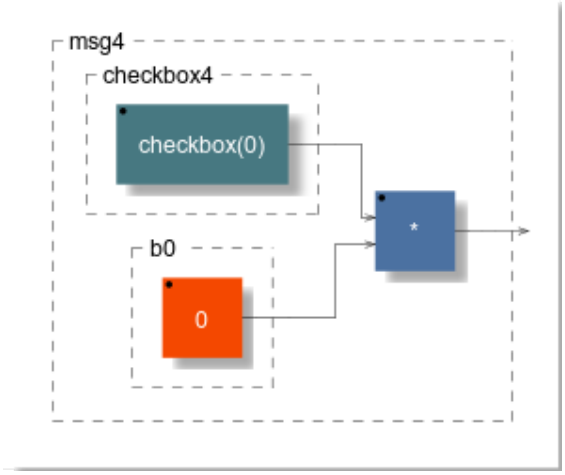

Figure 20: Equivalent of message object ”0”