Audio Engineering Society
Convention Paper 5756
Presented at the 114th Convention
2003 March 22–25 Amsterdam, The Netherlands

BlockCompiler: Efficient Simulation of Acoustic and Audio Systems

Matti Karjalainen
Helsinki Univ. of Tech., Lab. of Acoustics and Audio Signal Processing, P.O. Box 3000, FIN-02015 HUT, Finland
Correspondence should be addressed to Matti Karjalainen (`matti.karjalainen@hut.fi`)

ABSTRACT
An experimental software environment called the BlockCompiler is described that is developed for flexible yet efficient simulation of different acoustic and audio systems. It is based on computational block objects and their interconnection networks, and it supports several different modeling paradigms. It is particularly powerful in creating physical models where two-directional interaction between physical elements has to be represented. High-level model specifications are compiled to efficient code, supporting real-time simulation and sound synthesis of relatively complex systems. Simulation examples described include modeling of musical instruments, speech synthesis, and multidimensional acoustical structures. Further cases fitting to the scheme, such as loudspeaker simulation, are discussed briefly.

1 INTRODUCTION

Computer-based simulation of systems that include complex physical interactions is a tedious task, and often such simulations run slow unless special effort is made to optimize the code. Combination of different modeling paradigms makes it even more demanding, easily limiting the palette of applicable tools and techniques.

This work emerged from the need to easily write efficient simulation and sound synthesis programs for different kind of acoustic and audio applications using a multi-paradigm approach. The idea of block-based computation itself is well known; there is multitude of software environments such as Max/MSP and JMax [1, 2], Pd [3], LabView [4], Simulink [5], etc., that are based on wiring of DSP blocks, using a graphical interface or textual scripting. There are also software environments particularly for simulating distributed systems (FEM, BEM, FDTD) or lumped element systems (circuit simulators). None of them is, however, a truly multi-paradigm environment for flexible yet efficient simulation and real-time synthesis.

1.1 Overview of the BlockCompiler

The BlockCompiler is an experimental software environment using block objects and their interconnection networks for high-level specification of computational models. The first level of objects supports conventional DSP and one-directional signal data flow between objects. This includes elementary blocks such as adders, multipliers, nonlinear functions, filters, transformations, and sound I/O through PortAudio sound drivers. In addition to synchronous signal flow (through input and output ports), there is parametric control flow (through parameter ports) that supports asynchronous communication.

A more advanced level supports modeling of physical (two-way) interactions. The elements are connected through two-way ports that carry physical signal variables, such as force, pressure, velocity, voltage, current, etc. In this sense the system works as a circuit and network simulator. The structures may also be 2-D and 3-D meshes, which is important in simulating for example musical instruments or acoustic spaces. The models are based on Digital Waveguide (DWG) [6] and Finite Difference (FDTD) [7] principles. Lumped element systems can be added through Wave Digital Filter (WDF) [8] principles.

The most challenging modeling cases are those that are nonlinear or time-variant (non-LTI). Although there is no general theory how to deal with discrete-time simulation of them in a physically sound way, there are promising approaches that can guarantee for example the stability of the simulation even for nonlinear systems as far as the corresponding analog system is stable [9]. This topic is the most challenging direction for future work.

High-efficiency simulation in the BlockCompiler is based on optimized code that is produced from block-based description and compiled to run-time code. The high-level object-oriented part is written and scriptable in Common Lisp [10], which allows for high flexibility in manipulating computational structures. Scheduling, code generation, compilation, and starting to run medium-complexity scripts takes only few seconds, and the model can be edited and recompiled interactively and incrementally. A single model in BlockCompiler is built as a patch that can be inspected and controlled form Lisp level while the DSP loop is running. A patch can also be computed in a stepping mode.

So far the system does not have a graphic block diagram editor, because for an advanced user it is more productive to define models in textual form. For easy use by non-experts for constructing models from predefined modules it is, however, important to provide graphical programming, which is planned to be added in the future. The experimental prototype of the BlockCompiler runs only on Macintosh OS X operating system, but all its software components should be possible to be ported to other major platforms (Linux, Windows).

The paper describes BlockCompiler and particularly its applications to challenging audio and acoustics problems. Section 2 presents an overview of techniques to model spatially distributed and lumped systems, both by wave-based and Kirchhoff variable approaches, as well as their combinations to hybrid models. Section 3 describes the BlockCompiler with case studies on basic audio DSP. Section 4 describes physical modeling examples for real-time sound synthesis of musical instruments, speech synthesis by vocal-tract modeling, and multidimensional acoustical systems. Further possibilities, such as real-time loudspeaker simulation, are discussed briefly. Discussion and a summary are given in Section 6.

2 DISCRETE-TIME MODELING OF PHYSICAL INTERACTIONS

Physical modeling and model-based sound synthesis of musical instruments as well as of other sound sources have become an important part of computer music and modern audio [6]. Different approaches and modeling techniques have been proposed to describe and realize spatio-temporal physical behavior that is found for example in musical instruments. For efficient computation these models are formulated through DSP or equivalent algorithms to simulate physical interaction.

While in abstract synthesis by DSP algorithms the interaction and signal flow are mostly directed (one-directional), in more physical approaches, such as digital waveguides [6], wave digital filters [8, 9, 11], finite difference models [7, 12, 13], etc., the interaction is two-directional. In such cases we can make distinction between models with *Kirchhoff variables* (K-variables) and *wave variables* (W-variables). This distinction is based on the fact that in a Kirchhoff formulation the total observable variable is considered, while with wave variables the Kirchhoff variable is divided into one (or more) directed wave component pair(s).

As a continuation to studies in [14, 15, 16, 17], in this paper we are particularly interested to combine K-models and W-models in a coherent way and show how to implement them using the BlockCompiler. Finite difference time domain (FDTD) models are basically physical models where a Kirchhoff variable, such as force or pressure, is taken as a free physical variable and the complementary one (such as velocity or volume velocity) is determined through a parameter, such as impedance Z or its inverse, admittance $G = 1/Z$. The relation of FDTD models and wave-based models is discussed and the combination of such submodels is studied in particular. The utilization of wave digital filter (WDF) components for lumped elements is also discussed.

In a one-dimensional lossless medium the wave equation is written

$$y_{tt} = c^2 y_{xx} \tag{1}$$

where y is (any) wave variable, subscript tt refers to second partial derivative in time t, xx second partial derivative in place variable x, and c is speed of wavefront in the medium of interest. For example in a vibrating string we are primarily interested in transversal wave motion for which $c = \sqrt{T/\mu}$, where T is tension force and μ is mass per unit length of the string [18].

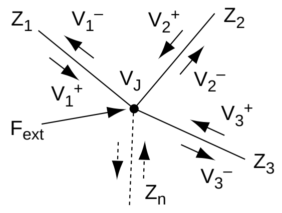


Fig. 1: Series junction of impedances Z_i with associated velocity waves indicated. A direct force input F_{ext} is also attached.

The two common forms of discretizing the wave equation for numerical simulation are through traveling wave solution and by finite difference formulation.

2.1 Wave-based modeling

The traveling wave formulation is based on the d'Alembert solution of propagation of two opposite direction waves, i.e.,

$$y(t,x) = \overrightarrow{y}(t - x/c) + \overleftarrow{y}(t + x/c) \tag{2}$$

where the arrows denote the right-going and the left-going components of the total waveform. Assuming that the signals are bandlimited to half of sampling rate, we may sample the traveling waves without losing any information by selecting T as the sample interval and X the position interval between samples so that $T = X/c$. Sampling is applied in a discrete time-space grid in which n and m are related to time and position, respectively. The discretized version of Eq. (2) becomes [6]:

$$y(n,m) = \overrightarrow{y}(n - m) + \overleftarrow{y}(n + m) \tag{3}$$

It follows that the wave propagation can be computed by updating state variables in two delay lines by

$$\overrightarrow{y}_{k,n+1} = \overrightarrow{y}_{k-1,n} \quad \text{and} \quad \overleftarrow{y}_{k,n+1} = \overleftarrow{y}_{k+1,n} \tag{4}$$

i.e., by simply shifting the samples to the right and left, respectively. This kind of discrete-time modeling is called Digital Waveguide (DWG) modeling [6].

The next step is to take into account the global physical constraints of continuity by Kirchhoff rules.

This means to formulate the scattering junctions of interconnected ports, with given impedances and wave variables at the related ports. Our goal is to end up with a systematic object-based representation for automatic compilation of block diagrams for physical models. For a scattering junction of Fig. 1, when the physical variables force F and velocity V are used and a series junction[1] impedance model of N ports is utilized [6], the Kirchhoff constrains are

$$V_1 = V_2 = \ldots = V_N = V_\mathrm{J} \tag{5}$$

$$F_1 + F_2 + \ldots + F_N = F_\mathrm{ext} \tag{6}$$

where V_J is the common velocity of coupled branches and F_ext is an external force to the junction. When port velocities are represented by incoming wave components V_i^+, outgoing wave components by V_i^-, impedances attached to each port by Z_i, and

$$V_i = V_i^+ + V_i^- \quad \text{and} \quad F_i^+ = Z_i V_i^+ \tag{7}$$

the force equation becomes

$$\begin{aligned} F_\mathrm{ext} &= \sum_{i=0}^{N-1} F_i = \sum_{i=0}^{N-1} (F_i^+ + F_i^-) \\ &= \sum_{i=0}^{N-1} (2 Z_i V_i^+ - Z_i V_\mathrm{J}) \end{aligned} \tag{8}$$

Solving for the junction velocity V_J yields:

$$V_\mathrm{J} = \frac{1}{Z_\mathrm{tot}} (F_\mathrm{ext} + 2 \sum_{i=0}^{N-1} Z_i V_i^+) \tag{9}$$

where $Z_\mathrm{tot} = \sum_{i=0}^{N-1} Z_i$ is the sum of all impedances to the junction. Outgoing velocity waves, obtained from Eq. (7), are then $V_i^- = V_\mathrm{J} - V_i^+$. The result is illustrated in Fig. 2. When impedances Z_i are frequency-dependent, this diagram can be interpreted as a filter structure where the incoming velocities are filtered by the corresponding wave impedances Z_i times two, and their sum is filtered further by $1/Z_\mathrm{tot}$ to get the junction velocity V_J. Displacement is obtained, for example in string modeling, by integrating the junction velocity V_J in time.

Two special cases can be mentioned based on Eq. (9). First, a (passive) loading impedance is the case with Z_i where no incoming velocity wave V_i^+ is associated. This needs no computation except including Z_i in Z_tot because the incoming wave is zero, see the left-hand termination in Fig. 2. Another factor is the external force F_ext effective to the junction. This is connected directly to the summation at the junction as shown in Fig. 2.

The wave variables and impedances at ports attached to a junction can be specified in any proper transform domain, but we are here interested in z-domain formulations for practical discrete-time computation. Notice that the impedances in Fig. 2 can be real-valued or frequency-dependent so that Z_i and the admittance $1/\sum Z_i$ can be realized as FIR or IIR filters, or just as real coefficients if all attached impedances are real. In the latter case, if we skip the external force F_ext of Eq. (9), we may write the equation using scattering parameters α_i as

$$V_\mathrm{J} = \sum_{i=0}^{N-1} \alpha_i V_i^+, \quad \text{where} \quad \alpha_i = 2 Z_i / Z_\mathrm{tot}. \tag{10}$$

This and other special forms of scattering are efficient computationally when impedances are real-valued, but in a general case it is practical to implement computation as shown in Fig. 2 so that the term $1/\sum Z_i$ is a common filter.

The freedom to use any impedance formulation allows also for applying measured data, such as bridge impedance/admittance as a part of an instrument model. The passivity condition is, as for a scattering junction in general, that all Z_i are positive real. Notice also that the realization of junction nodes as shown in Fig. 2 is general for any linear and time invariant system approximation, also for 2-D and 3-D mesh structures.

2.2 Finite difference modeling

In the most common way to discretize the wave equation by finite differences the partial derivatives in Eq. (1) are approximated by second order central finite differences

$$y_{xx} \approx (2 y_{x,t} - y_{x-\Delta x,t} - y_{x+\Delta x,t})/(\Delta x)^2 \tag{11}$$

$$y_{tt} \approx (2 y_{x,t} - y_{x,t-\Delta t} - y_{x,t+\Delta t})/(\Delta t)^2 \tag{12}$$

By selecting the discrete-time sampling interval Δt to correspond to spatial sampling interval Δx, i.e.,

[1]Parallel junctions and admittance models are not discussed for brevity.

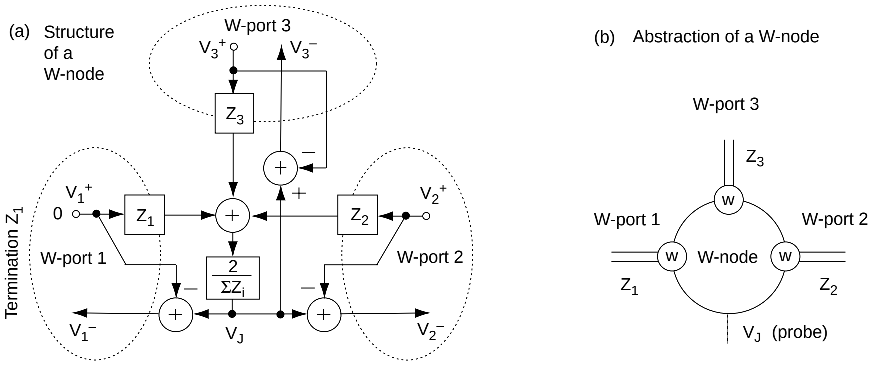


Fig. 2: N-port scattering junction (three ports are shown) of ports with impedances Z_i. Incoming velocities are V_i^+ and outgoing velocities V_i^-. W-port 1 is terminated by impedance Z_1 (notice zero incoming velocity).

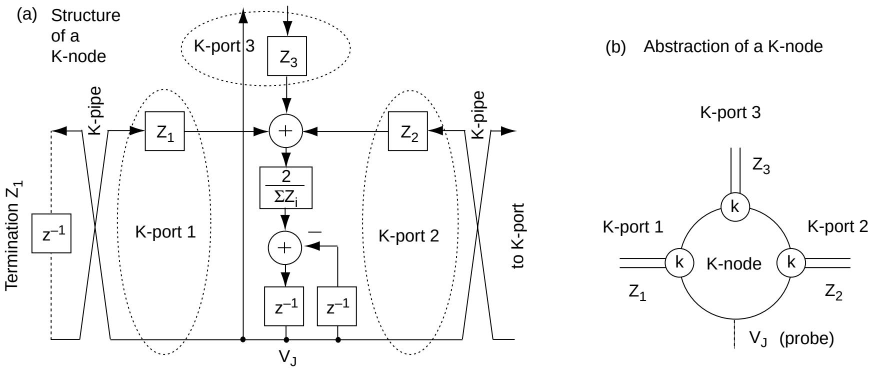


Fig. 3: Digital filter structure for finite difference approximation of a two-port scattering node with port impedances Z_1 and Z_2. K-port 1 is terminated by impedance Z_1. Only total velocity V_J (K-variable) is explicitly available. This functionally equivalent to the wave-based model of Fig. 2.

$\Delta t = c\Delta x$, and using index notation $k = x/\Delta x$ and $n = t/\Delta t$, Eqs. (11) and (12) result in

$$y_{k,n+1} = y_{k-1,n} + y_{k+1,n} - y_{k,n-1} \qquad (13)$$

which is a special case of multidimensional meshes as an FDTD formulation [6, 25]. From form (13) we can see that a new sample $y_{k,n+1}$ at position k and time index $n+1$ is computed as the sum of its neighboring position values minus the value at the position itself one sample period earlier.

The equivalence of digital waveguides and FDTDs [11], although being computationally different formulations, is also applicable to expand Eq. (13) to a scattering junction with arbitrary port impedances. Figure 3 depicts one scattering node of a 1-D FDTD waveguide and the way to terminate one port by impedance Z_1. There can be any number of ports attached also here as for a DWG junction.

An essential difference between DWGs of Fig. 2 and FDTDs of Fig. 3 is that while DWG junctions are connected through 2-directional delay lines (*W-lines*), FDTD nodes have two unit delays of internal memory and delay-free *K-pipes* connecting ports between nodes. These junction nodes and port types are thus not directly compatible (see next subsection). One further difference, in addition to algorithmic and computational precision properties, is the possibility of 'spurious' responses in FDTDs, i.e., an initial state of finite energy may generate waves of infinitely expanding energy in them [7]. This 'non-physical' behavior needs extra computation in DWGs to get a similar behavior.

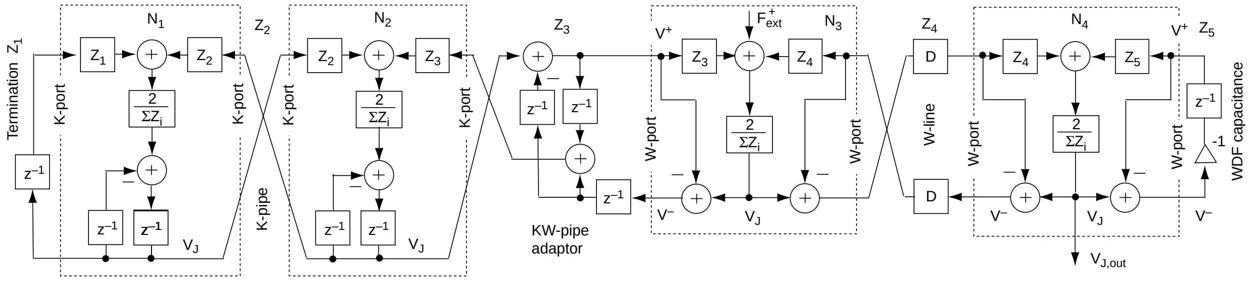


Fig. 4: FDTD elements (left) and DWG elements (right) forming a hybrid waveguide. Z_i are wave impedances of W-lines (for DWGs) and K-pipes (for FDTDs) between junction nodes (delimited by dashed lines). V_J are junction velocities, V^+ and V^- are wave components. Terminations: Z_1 (left) and WDF capacitor Z_5 (right). Delays marked by D may be of integer or fractional length. In: F^+_{ext}, Out: $V_{\text{J,out}}$.

2.3 Combining DWGs and FDTDs

The next question to discuss is the possibility to interface wave-based and FDTD-based submodels. In [16] it was shown how to interconnect a lossy 1-D FDTD waveguide with a similar DWG waveguide to form a hybrid model using a proper interconnection element (adaptor). In a similar way, it is possible to make an arbitrary hybrid model of K-elements (FDTD) and W-elements having arbitrary wave impedances at their ports.

Figure 4 shows how this is done in a one-dimensional modeling case, but each junction node might connect any number of related ports as well, making also 2-D and 3-D meshes possible. The left-hand sybsystem in Fig. 4 is an FDTD waveguide through ports of specified wave impedances, and the right-hand part is a similar formulation for a wave-based model (DWG).

The function of the KW-pipe in the middle of Fig. 4 between the FDTD node N_2 and DWG element N_3 is to adapt the K-type port of an FDTD node and the W-type port of a DWG node, and it is delay-free in the K-to-W direction. The proper functioning of the adaptor can be shown by testing the propagation of a left- and a right-traveling impulses through the adaptor. The equivalence of wave-based and K-variable based (FDTD) models and interfacing rules between them allow now for implementing mixed models where either one of the approaches can be selected according to which one is more useful in a problem at hand.

2.4 Including lumped and nonlinear elements

As noted above, any linear and time-invariant system (1-D, multidimensional, or irregularly structured collection) of properly connected blocks can be computed using the described formalism. A useful additional formalism is to adopt Wave Digital Filters (WDF) [8, 11] as discrete-time simulators of lumped parameter elements. Based on wave variables, they are computationally fully compatible with the structures described above.

A WDF *resistor* does not add much to the systems above, because it is just a special case of an arbitrary impedance termination discussed above, but WDF *capacitors* and *inductors*, as well as *ideal transformers* and *gyrators*, are useful components [8]. As a physically bound choice for the case of this study, a WDF capacitor is a feedback from V^- wave of a port back to V^+ through a unit delay and coefficient -1, and having a port impedance

$$Z_C = 1/2f_s C \qquad (14)$$

see the right-hand termination in Fig. 4. A WDF inductor is a feedback through a unit delay, having a port impedance

$$Z_L = 2f_s L \qquad (15)$$

Here C is capacitance, L is inductance, and f_s is the sample rate (cf. [11]). A beneficial property of these elements is, since their wave impedances are real-

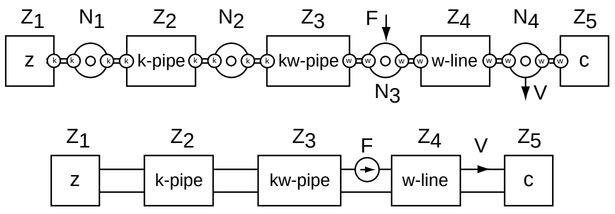


Fig. 5: Circuit type diagrams of the hybrid waveguide of Figure 4. Top: with explicit ports and series nodes ($\mathrm{N_i}$); bottom: more conventionally.

valued, that junctions of such ports remain memoryless in the sense of Fig. 2, i.e., Z_i and $1/\sum Z_i$ are real. On the other hand, arbitrary impedances, for example from measured data, provide high flexibility of design if junctions with memory are allowed.

The WDF formulation helps essentially in another problem that appears when nonlinearities or fast parametric changes in a system are to be modeled, where delay-free loops may appear, requiring special solutions. Simply inserting an extra delay makes the model non-physical and is a source of potential instability. Several solutions to this problem have been proposed, one of them being to connect a nonlinear component through a WDF adaptor so that a delay-free loop is eliminated and the structure has the same energetic behavior as the corresponding analog system [9]. Although possible in the BlockCompiler framework, this problem is however out of the scope of this paper.

2.5 Circuit formulation

A further step can be taken to make the formulations conceptually compatible with conventional circuit simulation. The diagram in Fig. 4 can be abstracted to a circuit model as a set of series connections of ports of the W- and K-type elements, i.e., K-pipe (Z2), adaptor KW-pipe (Z3), W-line (Z4), as well as impedances Z1 and Z5. The circuit is shown as a diagram in Fig. 5 by two abstractions.

3 THE BLOCK COMPILER

The development of the BlockCompiler started from the need of a flexible tool for efficient physical modeling, i.e., discrete-time simulation of physical systems, particularly acoustic and audio systems. In this section we describe the general properties of the BlockCompiler and demonstrate it through DSP-oriented examples. Section 4 shows how two-directional interaction principles, discussed above in Section 2, are realized and simulated in the BlockCompiler.

3.1 Software basis

As tools for the development of this experimental system, two programming languages were selected in order to provide both high flexibility and computational efficiency. Common Lisp [10] is known as the language for artificial intellegence where symbolic manipulation is of importance. With its object oriented extension CLOS (Common Lisp Object System) it is well suited to processing of expressions and object structures, which in this context means parsing of block-based computational specifications, scheduling them (i.e., ordering of operations) and generation of efficient code to be compiled to machine code. It is as well suitable to high-level parametric control of real-time processing. One advantage of the Lisp language over many alternatives is that it is available as an incremental compiler, that is, single expressions can be executed (compiled) independently, without need to compile entire files. This makes it useful as a scripting language as well.

Modern C compilers are know to generate highly optimized code for time-critical numerical computation. Thus the widely available C language was a natural choice for low-level compilation in the BlockCompiler. The role of Lisp is to generate C code from block-based specifications, and a C compiler compiles this to executable code. Thus the entire system combines the advantages and strongest features of both languages.

There are, of course, other alternatives of programming languages available to implement a similar functionality. C/C++ could be used easily to most parts of the compiler with the advantage of having only a single language, although some parts of symbolic manipulation are more tedious to program in it, and it is not an interactive scripting language. Combination of Java and C could be another alternative, as well as combination of modern scripting languages (such as Perl) and C for efficient code generation. The choice of Lisp and C was, however, well motivated in this case of experimental development. One disadvantage from the straightforward

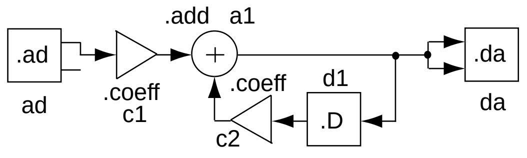


Fig. 6: Simple low-pass filtering from sound driver input to output.

use of Lisp is that inputs scripts for block structure specifications are in Lisp syntax, i.e., in parenthesis forms, which deviates from most other programming language syntaxes.

3.2 BlockCompiler features

The main implementation features and functional principles of the BlockCompiler are briefly described as follows:

• **Block structures and patches**: Block structures are composed of block objects for computational operations and interconnections between blocks for data flow and control flow. A full model specification is called a *patch*, which consists of interconnected *block-item* units. The principle of creating simple DSP patches is illustrated through the case of Fig. 6. A first-order recursive low-pass filter is constructed of coefficients **c1**, **c2**, adder **a1**, and delay **d1**. It is connected between sound driver input **ad** (AD-converter, one channel only) and output **da** (DA-converter, both channels of stereo).

This patch can be created by scripting:

```
(defparameter lpstream
   (patch ((ad (.ad))             ; sound in
           (c1 (.coeff 0.0666))   ; coeff
           (a (.add :inputs 2))   ; adder
           (c2 (.coeff -0.8668)) ; coeff
           (d1 (.D))              ; unit delay
           (da (.da)))            ; sound out
      (connect (output ad 0) (input c1))
      (connect (output c1) (input a 0))
      (connect (output a) (input d1))
      (connect (output d1) (input c2))
      (connect (output c2) (input a 1))
      (connect (output a) (input da 0))
      (connect (output a) (input da 1))))
```

or more compactly applying chaining function (->) to inputs and outputs in short-form notation:

```
(defparameter lpstream
  (patch ((a (.add)))
    (-> (.ad) (.coeff 0.0666) a (inputs (.da)))
    (-> a (.D) (.coeff -0.8668) (input a 1))))
```

In the forms above `defparameter` creates a global symbol of name following it (`lpstream` in this case), and the patch created by form `patch` is bound to this variable. The parenthesis forms following `patch` create the corresponding blocks, for example `(.ad)` creates the A/D conversion block and binds it to local variable `ad`. Symbol `.ad` is both the name of the A/D converter class and in the parenthesis form it is a make-function to instantiate an object of this class. (Comments follow the '`;`' character.)

After creating the block objects they are to be connected. The function `connect` is used to make connections between block outputs and inputs. This completes the low-pass filtering patch of Fig. 6.

The latter short-form specification is much more compact, since objects are created, when possible, as a part of interconnecting form while chaining of output to input, to output, etc., using the chaining function '`->`'. If there are more than one input or output for a block, inputs and outputs different from the first one (i.e., of index different from 0) must be expressed explicitly. (Notice that in general there can be any number of inputs and outputs to a block.)

When a patch object is created by evaluating a form such as shown above, it can be started streaming by calling (`run-patch lpstream`). Audio input is then continuously low-pass filtered to both audio output channels. The first-order low-pass filter above has a (3 dB) cutoff frequency of 1 kHz for sample rate of 44.1 kHz.

Notice that loops containing delays are allowed, but any delay-free loop in a structure will result in an error message in the scheduling phase when calling the `run-patch` or an equivalent function.

Notice also that it is more convenient to make DSP patches using specific built-in filter blocks than to construct them from most elementary blocks. These specialized DSP blocks are also typically more efficient due to more optimized code generation.

Inputs and outputs of a block are intended for synchronous data flow (see discussion on multirate computation below). In addition to inputs and outputs, a block can have parameter inputs that support also asynchronous communication so that a parameter input can be written when needed. Parameter inputs are referred to by form `(param block index)`.

• **Macro blocks** can be defined as combinations of more elementary blocks. This is a useful abstraction mechanism whereby the details of the the new class objects are hidden. The macro blocks can be utilized further as elements of new macro definitions. Parameters can be given to specify the properties of a macro block during instantiation.

A simple example of macro block definition, implementing the low-pass filter used in a patch related to Fig. 6, is:

```
(defclass .lpf (macro-block) ())
```

which defines a new block class named `.lpf` for the low-pass filter part of the patch. The instantiation of macro block `.lpf` is specified by method `make-macro-block` as:

```
(defmethod make-macro-block ((block .lpf) &key
                                  coeff1 coeff2)
  (let ((c1 (.coeff coeff1))
        (c2 (.coeff coeff2))
        (a (.add)))
    (-> c1 (input a 0) (.d) c2 (input a 1))
    (setf (inputs block) (list (input c1)))
    (setf (outputs block) (list (output a)))))
```

where keywords after `&key` are available to control the properties of the macro-block instances. Elementary blocks `c1` and `c2`, i.e., filter coefficients, as well as adder `a`, are specified and linked next. Finally, the inputs and outputs of the macro block are specified based on inputs and outputs of the elementary blocks[2].

The low-pass filtering patch of Fig. 6 can be made now using `.lpf` simply by

```
(defparameter lpstream
   (patch ()
      (-> (.ad)
          (.lpf :coeff1 0.0666 :coeff2 -0.8668)
          (inputs (.da)))))
```

[2]If parameter inputs are needed, they are included by form `(setf (params block) (list (param blockx index))))`.

• **Data types**: The BlockCompiler supports data types $\{short, long, float, double\}$, and corresponding array types are available as well. (Complex-valued data type will probably be added for a limited set of operations.) Data is normally transferred between blocks in port-related global variables or arrays (synchronous data-flow). This can be optimized further by using register variables. Memory allocation for blocks is carried out by the Lisp level, and compactness of the allocated memory is a goal to avoid cache conflicts while computing a patch.

• **Multirate support**: Multirate processing is available so that each block can be given a relative sample rate. (In the present version only down-sampling is supported by ratios $1/N$ where N is an integer.) This feature allows for multirate and polyphase DSP computation.

• **Scheduling**: A patch is scheduled by walking the (possibly hierarchical due to macro-blocks) structure and ordering the elementary operations. If there are delay-free loops or illegal structures, an error is reported. Automatic elimination or modification of delay-free loops is a challenging topic of future study.

• **Code generation and compilation**: Each block writes inline C code into a file to create a single function with related data and declarations. C code generation of each block class is defined in 'pseudo-C' which looks like C code but having data references to Lisp level. The resulting C file is then compiled by an automatic call to a C compiler.

• **Patch streaming and stepping**: The function pointer of the compiled code is taken and linked to the sample stream of the sound driver for real-time processing of `patch` by `(run-patch patch)`. While real-time streaming is running, the patch is fully controllable from Lisp, allowing for highly flexible control and inspection of the patch behavior.

A non-streaming way to utilize a patch is to call it in single steps. The step mode of `patch` is activated by `(use-patch patch)`, and then a single sample step is called by `(step-patch patch)`.

4 PHYSICAL MODELING USING THE BLOCK COMPILER

The main motivation for the development of the PatchCompiler was to make it a tool for physical modeling. As discussed in Section 2, a real physical model requires two-directional interaction between blocks, based either on K-ports or W-ports. In this section a set of cases are represented, starting from a semiphysical Karplus-Strong string model and proceeding to more directly physical cases.

4.1 Semiphysical models: Karplus-Strong

The Karplus-Strong model [19] is a classical example of modeling musical instruments, particularly string instruments, considered as an extremely reduced form of physical modeling.

Figure 7 shows a block diagram for a (modified) Karplus-Strong string model. It consists of a triggerable wavetable (noise burst) and a single (fractional) delay loop with a low-pass filter [20] to create the decaying string sound by a recursive comb filter[3]. The model of Fig. 7 can be realized with the following patch:

```
(defparameter ks
  (patch ((noise (qs::make-random-signal 10000.0
                   :span (qs::make-span 0 882)))
          (wt (.wtable :data (qs::s-array noise)))
          (dlx (.delay :delay-time 0.01
                       :control '.ap1))
          (add (.add :inputs 2))
          (lp (.lp1))
          (da (.da)))
    (-> wt add dlx lp (input add 1) (inputs da))))
```

where local variables are: `noise` keeps a random noise burst of 882 samples (20 ms), `wt` is the wavetable of the noise samples, `dlx` keeps the fractional delay with all-pass filter type of length control (initial length 10 ms), `add` is an adder, `lp` is a first-order low-pass for string loss control, and `da` is sound output. These elements can be interconnected by a single expression of input-output chaining.

When experimenting on synthesis models with parametric control it is important to be able to change the model parameters interactively. The following version of the Karplus-Strong patch includes a graphical user interface with slider controls for the main parameters: delay length, loop filter cutoff frequency parameter, and loop filter dc gain. Additionally the graphic control panel, as shown in Fig. 8, includes a button to start/stop running the patch and

[3]In the original K-S model the excitation was inserted as random numbers into the delay line and the loop filter was a simple two-tap FIR filter.

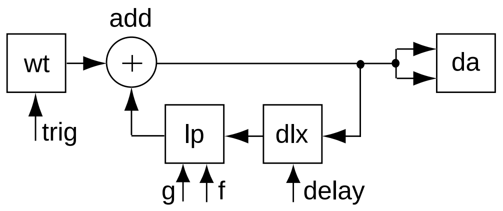
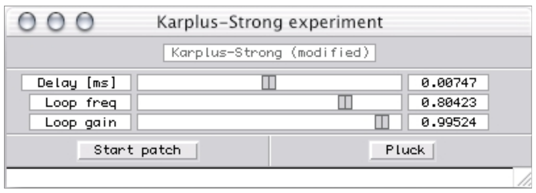


Fig. 7: Block diagram of a (modified) Karplus-Strong model as a case of semiphysical modeling.



Delay [ms]	0.00747
Loop freq	0.80423
Loop gain	0.99524

Fig. 8: Control dialog for the Karplus-Strong model.

trigger button for the wavetable to pluck the string model.

```
(defparameter ks
  (patch ((noise (qs::make-random-signal 10000.0
                   :span (qs::make-span 0 882)))
          (wt (.wtable :data (qs::s-array noise)))
          (dlx (.delay :delay-time 0.01
                       :control '.ap1))
          (add (.add :inputs 2))
          (lp (.lp1))
          (da (.da)))
    (-> wt add dlx lp (input add 1) (inputs da))))
    (make-controls
     (:v (text-box :text " Karplus-Strong (modified)")
         (make-instance 'qs::slider-box
           :view-subviews
           (list
            (lin-slider
              :title " Delay [ms]"   ;; slider
              :low 0.001 :high 0.014 ;; delay length
              :init-value 0.0075     ;; [seconds]
              :links (list (param dlx)))
            (lin-slider
              :title " Loop freq"    ;; slider
              :low 0.0 :high 1.0     ;; freq range
              :init-value 0.80       ;;
              :links (list (param lp 0)))
            (lin-slider
              :title " Loop gain"    ;; slider
              :low 0.9 :high 1.0     ;; gain range
              :init-value 0.995      ;;
              :links (list (param lp 1))))
           :length 200)
         (:h (start/stop-button
               :patch *current-patch*) ; start button
            (trig-button
              :text " Pluck"          ; trig button
              :links (list (param wt 0)))))
     :title "Karplus-Strong experiment"
     :view-position #@(300 100)
     :view-size #@(400 114))))
```

The sliders and the trigger buttons can be used to control the synthesis model when the patch has been activated from the `start-stop` button.

4.2 Physical modeling

In the modeling with physical interaction the blocks have ports of two-directional data flow. In Section 2 we studied K- and W-models using the example of Fig. 4. A patch to realize this model is given below. Both K-type and W-type nodes and lines/pipes are utilized, as well as an adaptor element (KW-line) to make a hybrid model. Terminations of the model are by impedance z1 on the left-hand side and by a WDF capacitor z5 on the right-hand side. Sound input (force excitation) is fed to node n3 through a `.source` injection element, and the velocity output of node n4 is fed to sound output (the A/D and D/A converters are not shown in Fig. 4).

```
(patch ((src (.source :port-type ‘K-port)) ;; force
        (n1 (.K-node)) (n2 (.K-node))  ;; nodes 1&2
        (n3 (.W-node)) (n4 (.W-node))  ;; nodes 3&4
        (k2 (.K-pipe :impedance 1.0))  ;; pipe Z2
        (kw (.KW-line :impedance 2.0)) ;; line Z3
        (w4 (.W-line :impedance 1.0 :length 10.5))
        (z1 (.Z :impedance 10.0 :port-type ‘K-port))
        (z5 (.C :capacitance 1.0e-3))) ;; WDF
  (connect (port k2 0) n1) (connect (port k2 1) n2)
  (connect (port kw 0) n2) (connect (port kw 1) n3)
  (connect (port w4 0) n3) (connect (port w4 1) n4)
  (connect (.ad) src) (connect src n3)
  (connect (port z1) n1) (connect (port z5) n4)
  (connect (probe n4) (inputs (.da))))   ;; output
```

Based on the circuit formulation in Fig. 5, a more compact and conceptually simple patch definition can be derived:

```
(patch ((src (.source :port-type ‘K-port)) ;; force
        (k2 (.K-pipe :impedance 1.0))  ;; pipe Z2
        (kw (.KW-line :impedance 2.0)) ;; line Z3
        (w4 (.W-line :impedance 1.0 :length 10.5))
        (z1 (.Z :impedance 10.0)) ;; termination
        (z5 (.C :capacitance 1.0e-3))) ;; WDF
  (-s z1 (port k2 0))
  (-s (port k2 1) (port kw 0))
  (-> (.ad) src)
  (-s (port kw 1) (port w4 0) src)
  (-> (probe (-s (port w4 1) z5)) (inputs (.da))))
```

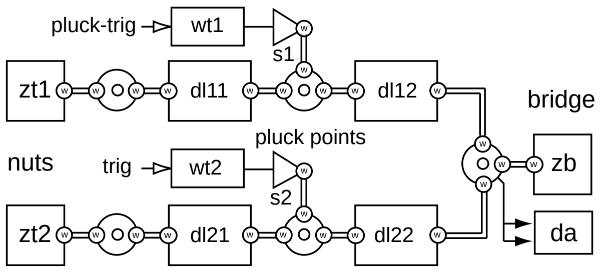


Fig. 9: Block diagram of two strings coupled through a common bridge impedance (Z_B).

where `-s` is a function for series connection of ports, resulting in a node of proper type. In the above forms `.source` is a block type that links force input to a series junction node and `probe` is a function that returns a probing output for the velocity of a node.

4.3 Plucked string modeling

Next we will explore how to build a string synthesis model, functionally similar to the Karplus-Strong model above but for two strings, using blocks with physical two-directional interaction. Figure 9 depicts a block diagram for two-strings with pluck wavetable inputs, strings being interconnected through a common bridge impedance, as well as bridge velocity output and nut end string terminations. A patch definition for the model is:

```
(defparameter strings2
   (patch ((noise (qs::make-random-signal 10000.0
                     :span (qs::make-span 0 300)))
           (znut :num ‘(1.195 -0.8)
                 :denom ‘(0.805 -0.8))
           (zbridge 100.0)
           (s1 (.source)) (s2 (.source))
           (zt1 (.z znut)) (zt2 (.z znut))
           (wt1 (.wtable :data noise))
           (dl11 (.d-line :delay-time 0.00510))
           (dl12 (.d-line :delay-time 0.00091))
           (wt2 (.wtable :data noise))
           (dl21 (.d-line :delay-time 0.00370))
           (dl22 (.d-line :delay-time 0.00080))
           (zb (.z zbridge))
           nb)
     (-s zt1 (port dl11 0))
     (-s (port dl11 1) (port dl12 0) s1)
     (-s zt2 (port dl21 0))
     (-s (port dl21 1) (port dl22 0) s2)
     (setq nb (-s (port dl12 1) (port dl22 1) zb))
     (-> wt1 s1) (-> wt2 s2)
     (-> (probe nb) (inputs (.da))))))
```

The strings are divided into two subsections so that the pluck excitation force can be injected from a triggerable wavetable (`wt1` and `wt2`) through force port (`s1` and `s2`) into a controllable plucking point (15 % from the bridge in this case). String 1 corresponds to 6th open string (82.4 Hz) and string 2 to 5th open string (110 Hz).

The string blocks (`dl11`, `dl12`, and `dl21`, `dl22`) are idealized (lossless, non-dispersive) transmission lines, and string losses are lumped to nut end termination impedances `zt1` and `zt2`, implementing frequency-dependent losses by IIR type of impedance specifications.

The common bridge impedance `zb` results in a sympathetic coupling between the strings. For simplicity `zb` has here a frequency-independent value[4], but it can be given any specification of FIR or IIR type, including a measured or modeled impedance of relatively high high order.

The output signal is probed from the bridge velocity. It could be processed further through a body filter in order to simulate the sound radiation of an acoustic guitar through a body, but again for simplicity it is fed directly to sound output.

To make the model more realistic for example for the acoustic guitar, there should be six strings and each one should have dual-polarizations, i.e., two submodels. Furthermore, finger-string interaction could be simulated by a fingertip model with varying parameters, based on contact state of the finger and the string.

4.4 Vocal tract modeling

Transmission line modeling of the vocal tract is a traditional physical way to simulate speech production. Although modern speech synthesis techniques don't use such an approach in practice, physical vocal tract modeling gives the best understanding of the underlying phenomena.

The Kelly-Lochbaum transmission line model [21] consist of two-directional delay lines and scattering junctions to simulate the wave behavior in a tube of cross-sectional area varying along the tract. Terminations at the glottis and lips are realized by proper impedances. The glottal source can be implemented as a nonlinear oscillating vibroacoustic subsystem, but for simplicity the glottal pulse is often injected as a signal from a properly designed oscillator, such as the Klatt glottal model [22].

[4]All impedances are relative to string impedance which has normalized value of 1.0.

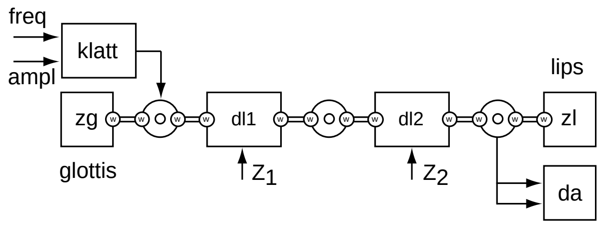


Fig. 10: Block diagram of speech synthesis by transmission-line vocal tract modeling.

A block diagram of such a model with two vocal tract subsections is shown in Fig. 10. A patch to realize it is:

```
(defparameter speech
   (patch ((gl (.glottis-klatt))
           (zg (.Z 50.0)) (s (.source))
           (dl1 (.delay :delay-time 0.0025
                    :impedance 3.0))
           (dl2 (.delay :delay-time 0.0025
                    :impedance 0.3))
           (zl (.Z :num '(0.8 -0.8)
                    :denom '(1.2 -0.8))))
     (-> gl s) (-s s zg (port dl1 0))
     (-s (port dl1 1) (port dl2 0))
     (-> (probe (-s (port dl2 1) zl))
         (inputs (.da)))))
```

The vocal tract with total propagation delay of 0.5 ms is divided into a front section `dl1` and a back section `dl2`. The acoustic impedance in each section can be controlled according to formula

$$Z_{\mathrm{ac}} = \rho c/A \tag{16}$$

where ρ is the density of air, c the sound velocity, and A is the cross-sectional area of the tract. The acoustic impedances can be controlled dynamically through parameter inputs `z1` and `z2` of the delay lines. Some vowels, such as Finnish /a/, /y/, and the neutral vowel with homogeneous profile can be simulated by this two-section model, but a higher number of controllable sections are needed for more general synthesis in order to approximate arbitrary vocal tract shapes. (The values in the patch are for an /a/-like vowel.)

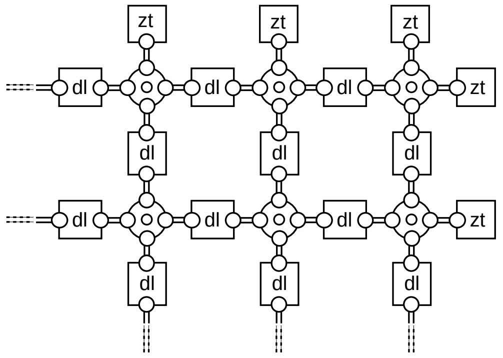


Fig. 11: Part of a two-dimensional rectangular digital waveguide mesh for membrane simulation, composed of delay lines `dl` and series nodes `o`, and terminating impedances `zt` at boundaries.

4.5 2-D modeling: waveguide mesh

The formalisms introduced above are general in the sense that they expand easily to modeling of 2-D and 3-D systems, such as waveguide meshes [23, 24, 25, 11]. The scattering junctions (nodes) allow for connecting two-port elements for example to rectangularly organized waveguide meshes, but are suitable to any regular or irregular structures.

As an example of a 2-D structure, a corner portion of a rectangular digital waveguide mesh is shown in Fig. 11. Both W- and K-modeling as well as their hybrids can be applied (W-modeling is used in this case). In FDTD modeling with K-variables the advantage is that only two unit delays (memory positions) are needed per node (see Fig. 3) both in 2-D and 3-D cases, while with DWGs (Fig. 2) four unit delays are required in 2-D modeling and six for 3-D modeling per node. Connecting the driving force is easier with DWGs while with FDTDs care should be taken to avoid spurious oscillations.

The mesh of Fig. 11 is terminated by impedances `zt` at the boundaries. The delay-lines `dl` may have identical or different impedances. In the latter case we can simulate media where the wave impedance varies spatially.

The digital waveguide mesh of Fig. 11 can be used for example to simulate a rectangular[5] membrane that is fixed at boundaries. The following script characterizes an implementation of the membrane model in the BlockCompiler.

```
(defparameter mesh
   (patch ((n 20) (m 20)   ; dimensions
          (arr (make-node-array n m))     ; nodes
          (wt (.wtable :data '(10000.0))) ; impulse
          (src (.source))
          (nsrc 14) (msrc 14)  ; input point
          (nout 5) (mout 5)    ; outout point
          (zt '3000.0)    ; termination impedance
    (terminate-node-array arr n m zt)
    (connect-node-array arr n m)
    (-> wt src)
    (-s src (cl:aref arr nsrc msrc))
    (-> (probe (cl:aref arr nout mout))
        (inputs (.da)))
    (defun hit () (trigger wt))
```

In this patch `n` and `m` are dimensions of the mesh and 2-D array `arr` holds the nodes of the mesh. The mesh is terminated by boundary impedances through function `terminate-node-array` and connected by `connect-node-array`. (Details of these functions are omitted here.)

An impulse excitation wavetable is connected to node (14,14) and output probe is at node (5,5). Since the spatial and the temporal sampling are bound by $\Delta T = \Delta X / c$, where c is wave propagation velocity, and because the number of elements is limited in practice by processor power, a decimated sample rate must be used in real-time computation. For a 1 GHz PowerPC processor the upper bound of mesh element count in the model of this case is about 20x20 for a 44100 Hz sample rate. When decimating the sample rate, a membrane with larger dimensions can be computed in real time. If the wave impedances can be identical, an essentially more efficient membrane model can be made using the FDTD approach and optimized scattering junctions.

The 2-D case can be easily extended to 3-D for example to simulate room acoustics at low frequencies [25], although real-time simulation is not practical anymore.

4.6 Loudspeaker equivalent circuit

The combination of DWG, FDTD, and WDF techniques provides a strong starting point for time-domain modeling of physical systems. We are currently expanding the BlockCompiler to circuit simulation with lumped elements. This is important

[5]By proper indexing the terminations can be positioned as well (approximately) circularly or by another boundary geometry.

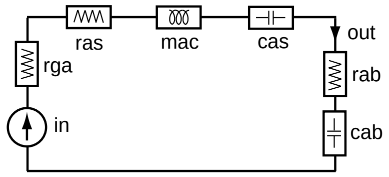


Fig. 12: Equivalent circuit of a closed-box loudspeaker.

not only in electrical circuit simulation but also in mechanical and acoustical modeling. A such case is time-domain simulation of loudspeaker behavior.

Wave digital filters are an inherently suitable technique to lumped element circuit simulation, although there is the problem of frequency scale warping due to bilinear mapping [8]. Here we want to see the applicability of DWG-type impedances in circuit simulation. As an example, the equivalent circuit of a closed-box loudspeaker [26], shown in Fig. 12, is realizable in BlockCompiler in the form:

```
(defparameter lspeaker
   (patch ((rga (.R *rga-spec*))
           (ras (.R *ras-spec*))
           (mac (.Z *dmac-spec*))
           (cas (.Z *cas-value*))
           (rab (.Z *rab-value*))
           (cab (.Z *cab-value*))
           (src (.source)) n)
     (-> (.ad) src)
     (setq n (-s src rga mac cas rab cab))
     (-> (probe n) (inputs (.da)))))
```

where component behaviors can be given in the form of FIR of IIR expressions in the global variables denoted `*xxx*`. The model is built using series connection function (`-s`) as well as with linking to sound driver input and output for real-time simulation. The `probe` function gets the volume velocity output of the model. Mapping to radiated far-field pressure should be realized as an additional element.

5 CONTROL PROTOCOL IN XML

In scriptable block-based systems, including the BlockCompiler, there is a tendency to use the syntax of the underlying programming language as a syntax for model scripting. It would be beneficial instead to develop carefully designed model libraries with portable and extendable syntax to specify and control models, being retargetable to different implementations of modeling software. XML (Extensible Markup Language) [27] is a particularly potential basis for such a protocol.

There is not yet such an extension to BlockCompiler, but to describe the idea, the patch in Fig. 6 could be expressed in XML form for example in the following way:

```
<?xml version="1.0"?>
<patch name="lpstream">
   <make-blocks>
      <ad-in block="ad1"/>
      <coeff block="c1" value="0.0666"/>
      <adder block="a1"/>
      <coeff block="c2" value="-0.8668"/>
      <unit-delay block="d1"/>
      <da-out block="da1"/>
   </make-blocks>
   <chain>
      ad1 c1 a1 <inputs block="da1"/>
   </chain>
   <chain>
      a1 d1 c2 <input block="a1" index="1"/>
   </chain>
</patch>
```

When a block model has been instantiated, it could be controlled by sending parameter messages to blocks also using XML syntax.

6 DISCUSSION AND SUMMARY

In this paper a block-based formulation of a computational environment for the simulation of physical and DSP systems has been presented, called the BlockCompiler. At this moment the system is highly experimental, and changes to the details of realization are expected in order to make it more systematic and general. The aim of this paper has been to describe the general approach as well as to demonstrate the flexibility and efficiency of the tool in simulating a wide variety of physical and signal processing systems.

The BlockCompiler prototype system works presently only on the Macintosh OS X operating system, but all software components (Lisp, C compiler, PortAudio sound driver) are available for other major platforms as well. The system has been found highly interactive due to simple scripting of model definitions and fast compilation to efficient run-time code. It is a flexible tool for basic research and application development. Simulation examples and further information on the system are available at 'www.acoustics.hut.fi/software/BlockCompiler'.

7 ACKNOWLEDGMENTS

This work is supported by the IST/ALMA project (ALgorithms for the Modelling of Acoustic interactions, IST-2000-30072), and Academy of Finland project 'Technology for Audio and Speech Processing' (53537). Special thanks are due to Cumhur Erkut and Lauri Savioja for helping in figuring out relations between DWG and FDTD formalisms.

8 REFERENCES

[1] Max/MSP, software by Cycling '74, `http://www.cycling74.com/products/maxmsp.html`

[2] jMax, software from IRCAM, `http://www.ircam.fr/produits/logiciels/ jmax-e.html`

[3] Pd, software by Miller Puckett, `http://www.crca.ucsd.edu/~msp/software.html`

[4] LabView, software by National Instruments Co., `http://www.ni.com/`

[5] Simulink, software by MathWorks, `http://www.mathworks.com`

[6] J. O. Smith, "Principles of Waveguide Models of Musical Instruments," in *Applications of Digital Signal Processing to Audio and Acoustics*, ed. M. Kahrs and K. Brandenburg, Kluwer Academic Publishers, Boston 1998.

[7] J. Strikverda, *Finite Difference Schemes and Partial Differential Equations*, Wadsworth and Brooks, Grove, Ca, 1989.

[8] A. Fettweis, "Wave Digital Filters: Theory and Practice," *Proc. IEEE*, 74(2), pp. 270–372, 1986.

[9] A. Sarti and G. De Poli, "Toward Nonlinear Wave Digital Filters," *Proc. IEEE Trans. Signal Processing*, vol. 47, no. 6, pp. 1654–1668, June 1999.

[10] G. L. Steele, *Common Lisp, The Language.* 2nd Edition, Digital Press, 1990.

[11] S. D. Bilbao, *Wave and Scattering Methods for the Numerical Integration of Partial Differential Equations*, PhD Thesis, Stanford University, May 2001.

[12] L. Hiller and P. Ruiz, "Synthesizing Musical Sounds by Solving the Wave Equation for Vibrating Objects: Part I, Part II," *J. Audio Eng. Soc.* vol. 19, nr. 6 and nr. 7, pp. 452–470 and 542–551, 1971.

[13] A. Chaigne, "On the use of finite differences for musical synthesis. Application to plucked stringed instruments", *J. Acoustique,* vol. 5, pp. 181–211, April 1992.

[14] M. Karjalainen, "1-D digital waveguide modeling for improved sound synthesis," *Proc. IEEE ICASSP'2001*, pp. 1869–1872, Orlando, 2002.

[15] C. Erkut and M. Karjalainen, "Virtual strings based on a 1-D FDTD waveguide model," *Proc. AES 22nd Int. Conf.*, pp. 317–323, Espoo, Finland, 2002.

[16] C. Erkut and M. Karjalainen, "Finite Difference Method vs. Digital Waveguide Method in String Instrument Modeling and Synthesis," *Proc. ISMA'2002*, Mexico City, Dec. 2002.

[17] M. Karjalainen, C. Erkut, and L. Savioja, "Compilation of Unified Physical Models for Efficient Sound Synthesis," submitted to *IEEE ICASSP'2003.*

[18] N. H. Fletcher and T. D. Rossing, *The Physics of Musical Instruments*, Springer-Verlag, New York, 1991.

[19] K. Karplus and A. Strong, "Digital Synthesis of Plucked-String and Drum Timbres," *Computer Music J.*, vol. 7, nr. 2, pp. 43–55, 1983.

[20] M. Karjalainen, V. Välimäki, and T. Tolonen, "Plucked String Models: from Karplus-Strong Algorithm to Digital Waveguides and Beyond," *Computer Music J.*, vol. 22, no. 3, pp. 17-32, 1998.

[21] J. L. Kelly and C. C. Lochbaum, "Speech Synthesis," *Proc. Fourth Int. Congr. Acoust.,* Copenhagen, Denmark, 1962, Paper G42, pp. 1-4.

[22] D. H. Klatt, "Software for a Cascade/Parallel Formant Synthesizer," *J. Acoust. Soc. Am.* **67**, pp. 971–995.

[23] S. Van Duyne and J. O. Smith, "Physical Modeling with the 2-D Digital Waveguide Mesh," *Proc. Int. Computer Music Conf. (ICMC'93).* Tokio, Japan, 1993, pp. 40–47.

[24] L. Savioja, T. J. Rinne, and T. Takala, "Simulation of Room Acoustics with a 3-D Finite Difference Mesh," *Proc. Int. Comp. Music. Conf. (ICMC 1994)*, pp. 463-466, Aarhus, Denmark, 1994.

[25] L. Savioja, *Modeling Techniques for Virtual Acoustics.* PhD thesis, Helsinki Univ. of Tech., Espoo, Finland, 1999.

[26] R. H. Small, "Closed-Box Loudspeaker Systems, Part I: Analysis," *J. Audio Eng. Soc.*, vol. 20, no. 10, 1972, pp. 798–808.

[27] XML documentation, in `http://www.w3.org/XML/`