Modalys-ER for OpenMusic (MfOM): virtual instruments and virtual musicians

RICHARD POLFREMAN

Music Department, University of Hertfordshire, College Lane, Hatfield, Hertfordshire AL10 9AB
E-mail: r.p.polfreman@herts.ac.uk

Modalys-ER is a graphical environment for creating physical model instruments and generating musical sounds with them. While Modalys-ER provides users with a relatively simple-to-use interface, it has only limited methods for mapping control data onto model parameters for performance. While these are sufficient for many interesting applications, they do not bridge the gap from high-level specifications such as MIDI files or Standard Western Notation (SWN) down to low-level parameters within the physical model. With this issue in mind, a part of Modalys-ER has now been ported to OpenMusic, providing a platform for developing more sophisticated automation and control systems that can be specified through OpenMusic's visual programming interface. An overview of the MfOM library is presented and illustrated with several musical examples using some early mapping designs. Also, some of the issues relating to building and controlling virtual instruments are discussed and future directions for research in this area are suggested. The first release is now available via the IRCAM Software Forum.

1. INTRODUCTION

Software sound synthesis typically involves the configuration of some combination of units that together form a system for generating sequences of sample values. Direct specification of sample data would not only be incredibly cumbersome (e.g. defining 44,100 16-bit values per audio channel per second for CD-quality sound), but also, other than for simple periodic waveforms, there is no readily perceivable connection between the sample values and the sonic result. Solutions to this problem are many and varied. Signal-based models have been historically dominant, using techniques such as additive and subtractive synthesis, amplitude and frequency modulation, and waveshaping. More recently, physical modelling (PM) techniques have become increasingly popular for producing sounds more acoustically plausible than those achieved via signal models. In either approach, there are usually many parameters to be specified (although techniques such as FM can produce rich timbres with relatively few), from initial configuration of the synthesis components (e.g. a signal path or combination of objects and interactions in a physical model) through to dynamically variable control values.

Today, sound synthesis performs many different roles: as a replacement for acoustic instruments, for generating 'synthetic sounds' within an otherwise traditional music context, for creating sound effects in film, television and computer games, as a source of abstract timbres in acousmatic music and sound art, etc. Software synthesis tools can be directed towards particular fields, taking advantage of users' domain expertise, rather than being left more open and typically requiring high levels of application-specific technical knowledge in order to utilise them effectively. An example can be found in Propellerheads' Reason™ software, which provides a system heavily modelled on standard MIDI hardware based studios, using a virtual rack of devices (such as a mixer, samplers, analogue synthesizers), with interconnecting cables and user interfaces consisting of dials, buttons and small displays. Most of the devices are preconfigured to a large extent as with their hardware counterparts. Anyone familiar with current project studios can then readily apply their expertise to this system.

This article is concerned with PM synthesis and how this can be used effectively by composers. Modalys (Morrison and Adrien 1993) is a PM synthesis program developed at IRCAM using modal modelling. This involves the simulation of the vibration patterns in systems of interacting virtual objects (such as plates, strings and tubes) in order to produce sound. One important feature of the modal synthesis engine is that users can define their own instruments with arbitrary numbers of simple components described in physical terms, whereas most waveguide systems available do not present such flexibility in a simple way. The main cost of the modal approach is in terms of processing speed – the engine is not yet real-time and complex instruments can take some time to synthesise. The Modalys user interface takes the form of a Scheme (a Lisp dialect) interpreter and the user must write a series of instructions in order to create objects and establish connections between them. There is little pre-defined structure above the object-interaction level, although with use of object-oriented programming techniques, higher-level entities can be defined and manipulated. As such, Modalys is a very open system, but can be awkward to utilise effectively in a musical context.

Developed by the author in collaboration with

Organised Sound **7**(3): 325–338 © 2002 Cambridge University Press. Printed in the United Kingdom. DOI:10.1017/S1355771802003126

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

IRCAM, Modalys-ER (Polfreman 1999) attempts to provide a relatively simple-to-use environment for designing and controlling physical model sound synthesis. Employing the same synthesis engine as Modalys, Modalys-ER gives users a graphical user interface and provides some high-level organisation lacking in Modalys. The original system, Modalyser, was a separate application that wrote Scheme files to be loaded into Modalys, while Modalys-ER integrates the synthesis functions directly into a single application. Recent versions (Modalys-ER 2.1 is the latest at the time of writing) have upgraded the visual representations used, simplified the *techniques* specification, added an 'Instrument Wizard' for aiding instrument construction, and provided a system for plotting interaction behaviours for debugging a synthesis.

A synthesis in Modalys-ER is specified using an *instrument* and a *score*, a deliberately familiar division separating a core synthesis specification from a set of time-varying parameters applied at synthesis time. The development of Modalys-ER was informed by wider research involving task analysis (Johnson 1992) of music composition (Polfreman 1999, Polfreman and Loomes 2001). This work attempted to define a generic model of the process of music composition, including descriptions of subtasks, their interrelations and the tools used (both in composition and for performance of a work). The separation of instrument (from performer) from score was used in this model as being widely applicable to both acoustic and electronic/computer-based situations. The role of the performer is designated as interpreting a set of instructions (a score) into a series of applied techniques involving an instrument, thus giving a four-level model: score – performer – technique – instrument. In many computer music systems, what is referred to as a 'score' may be considered as actually the output of a performer in our model, since it is a direct specification of instrument control values. (This in fact applies to Modalys-ER since the 'score' is at the performer level rather than a true score level.) Much of this paper refers to a simplified two-level structure of instrument and score, but we return to the role of performers later. While one can imagine a synthesis system without this type of separation (for example in a Max/MSP patch), it can be regarded as simply a convenient abstraction of all parameters that are likely to be changed over (synthesis) time into a separate area, leaving the central configuration static. The composer can then concentrate on working with the evolution of control values and be no longer concerned with much of the internal details of the synthesis process.

Modalys-ER adopted the instrument-score approach for two main reasons. First, the PM paradigm of Modalys provides a computer-based analogy of acoustic instruments, and so the score–instrument model can aid users by extending that analogy. Secondly, the design of instruments can be quite challenging and score–instrument separation allows re-use of existing instruments – particularly useful for those users not wishing to involve themselves in the technical details of instrument definition (re-use is also a more general aid to usability). The recent addition of an 'Instrument Wizard' takes advantage of this separation, by providing automatic construction of instruments from a few user-defined parameters which the composer can then utilise without much detailed understanding of the instrument configuration. Re-use should also apply in the opposite direction: a given score (representing a particular musical idea or gesture) should be capable of application to different instruments. The low-level scoring system used in Modalys-ER makes this relatively difficult at this time, since the score envelope shapes required to invoke a particular excitation (moving a plectrum, pushing a bow) are too closely associated with the physical excitation mechanism. In future developments it is hoped to resolve this issue.

Instrument–score separation has been used previously in well-known software synthesis applications such as Csound (Vercoe 1993), as well as forming the foundation of MIDI. While the instrument–score approach may be seen as simply following the traditional acoustic music path, there is a variable division of labour between the two (although perhaps not as much as in a system such as Csound). For example, a Modalys-ER instrument can be designed with no score controllable parameters, i.e. the instrument is self-contained and all key synthesis parameters are defined within it. In this case, a score is simply a tool to load the instrument, specify the duration of the synthesis and commence processing – making the score–instrument separation mainly redundant. Alternatively, many instrument variables may be mapped to score parameters, giving the score wide-ranging control over the physical interactions that occur during a synthesis. There are also different levels at which score control can be applied. For example, it may specify directly the position of plectrum over time in order to cross and re-cross a string, or, a sine controller may be used to achieve this motion and the score given control over the frequency of this controller. In MfOM, the score–instrument separation is important since it is the mechanism by which internal parameters of the instrument are exposed to the rest of the environment and so can be algorithmically derived. The new OM Modalys 2.0 library (see section 5) provides an alternative format for Modalys synthesis in OpenMusic without necessarily using an instrument–score separation.

While a Modalys-ER score can be seen as simply a collection of envelopes, the instrument is further divided into *construction* and *techniques*. The construction defines the make-up of the instrument in terms of *objects* (plates, strings, masses, etc.) and *connections* (such as force, pluck, strike) that are linked together in a patch. Parameters within the construction defined as *dynamic*

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

can then be accessed by the techniques, each of which maps an envelope control from the score onto one or more dynamic controls in the construction. The main purposes of this mapping system are: to allow a single score value to control more than one instrument parameter; to allow parameter mappings to change according to other parameter values (e.g. changing a pitch value may cause an excitation technique to move from driving a mass plucking one string to a mass on another string); to enable switching of techniques (e.g. have an instrument change from bowing to plucking); and to allow score reuse with a range of instruments – which as stated earlier has not yet been properly realised.

Figure 1 shows a Modalys-ER instrument and figure 2 a score for controlling it. In the instrument a *mass* is used as a reed and is linked via a *reed* connection (which simulates the mouthpiece and breath control) to a tube. Two *pickup* connections made to the tube are used to produce the output sound, while a *hole* connection allows us to open and close a hole in the tube. The additional connection to the mass is a *position* connection that is needed in order to clamp one end of the reed in place.

Two parameters in the construction have been made dynamic and appear at the top of the techniques area – the hole radius and the breath pressure. These are then mapped to the techniques *pitch* and *breath* respectively, whose behaviours can be dependent on individual pitch ranges. The breath technique uses a single pitch range (i.e. is independent of pitch), while the pitch technique uses two ranges – in one of which it opens the hole and in the other closes it. Each technique is then controlled by its own envelope in the score. The grey line across the pitch envelope indicates the pitch value at which the hole switches from closed to open.

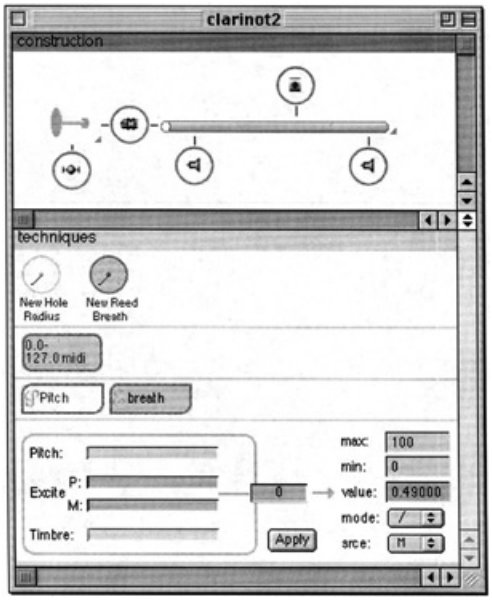


Figure 1. A Modalys-ER reed instrument with construction at the top, techniques at the bottom.

OpenMusic (Assayag, Rueda, Laurson, Agon and Delerue 1999) is a visual programming environment built on top of Digitool's Macintosh Common Lisp (MCL), using graphical entities for representing programming concepts such as classes, instances and functions. While OpenMusic can be used purely for writing object-oriented Lisp (CLOS) programs, music-specific libraries are provided for generating, editing and manipulating musical information. Programmers can also create their own libraries that can be added to the system. While in the past the system has been much used for algorithmic composition of notated music, the more recent addition of the omChroma library (Agon, Stroppa and Assayag 2000) has enabled effective control of sound synthesis through the generation of parameters that are then delivered to an external software synthesizer. This is typically carried out by sending system messages and/or text files to another application (e.g. Csound or IRCAM's Chant library). OpenMusic uses a patch notation for representing a program, with interconnected functions and class instances forming an algorithm that when evaluated produces some result – for synthesis usually a sound file. Figure 3 shows an omChroma patch generating sound using Csound. The central box displaying envelopes is an event class for a given Csound instrument. The instrument in this case is a simple oscillator that uses table f1 (here a sine wave) for the oscillator waveform and f11 (an exponential decay) for the amplitude envelope. The event class is given lists of amplitude, frequency and duration values and will generate a single note layering partials from the different values on top of each other, here giving a bell-like sound.

2. MfOM OVERVIEW

MfOM is a new OpenMusic library for generating sounds with physical model synthesis using the same synthesis engine as Modalys-ER and Modalys. Since Modalys-ER was developed using MCL, some of the code was directly ported to OpenMusic, although many important changes were necessary in order to successfully integrate Modalys-ER functionality within the OpenMusic structure. In particular, various technical issues had to be solved in order to provide stable integration of the Modalys synthesis engine, which is loaded as a shared library.

In MfOM, the class *ModInst* represents a Modalys-ER instrument. A ModInst instance (or a factory – a box that instantiates objects of a given class) can be opened by double-click and edited in a similar way to a Modalys-ER instrument, the difference being that there is no techniques area, only construction. External control of the instrument is again by means of assigning dynamic controls within the construction. Once an instrument has been built, it must be connected to the

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

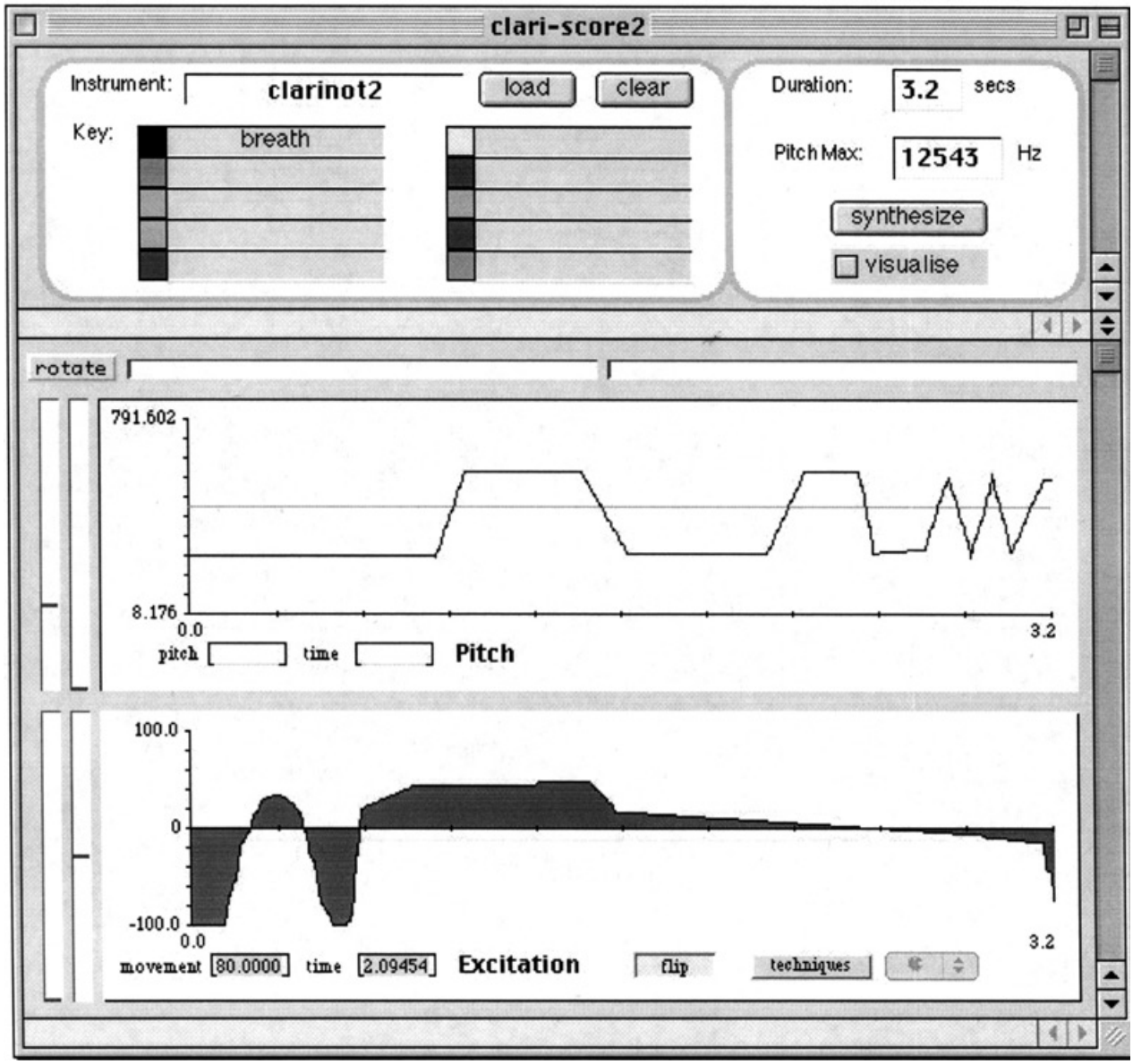


Figure 2. A Modalys-ER score for controlling the reed instrument shown in figure 1.

function *get-modalyser-instrument*, which when evaluated defines a new synthesis event class for that instrument, whose name is of the form *MO-EVT-instrumentName*. This new class is similar in behaviour to other omChroma classes for controlling Csound and other synthesis systems. It can be thought of as containing a copy of the instrument used to generate it, and also a set of inputs that specify values for the dynamic controls over synthesis time. An instance of the event class (or the class factory) can be opened, giving the user editable envelopes for each dynamic parameter. However, using these directly does not achieve any advance in control over that of Modalys-ER. The real gain is by using OpenMusic classes and functions to generate these values in some way and connect them to the inputs of the event to control it. In doing this it is possible to build systems for mapping high-level representations onto the synthesis control.

As stated earlier, the role of translating a score into a series of (parallel) physical actions involving an instrument is undertaken by a performer or musician, thus here the attempt is to construct some form of 'virtual musician' in order to play our 'virtual instrument'. SWN can be seen as a relatively poor representation for controlling instruments directly simply because it has evolved with a reliance on musicians' interpretation, but since SWN remains such a pervasive visual representation for music, it is useful to provide such mappings.

Once an event class has been defined and control values set, it is then connected to the generic function *synthesize* which is also given various global parameters for the synthesis – duration, number of audio channels,

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

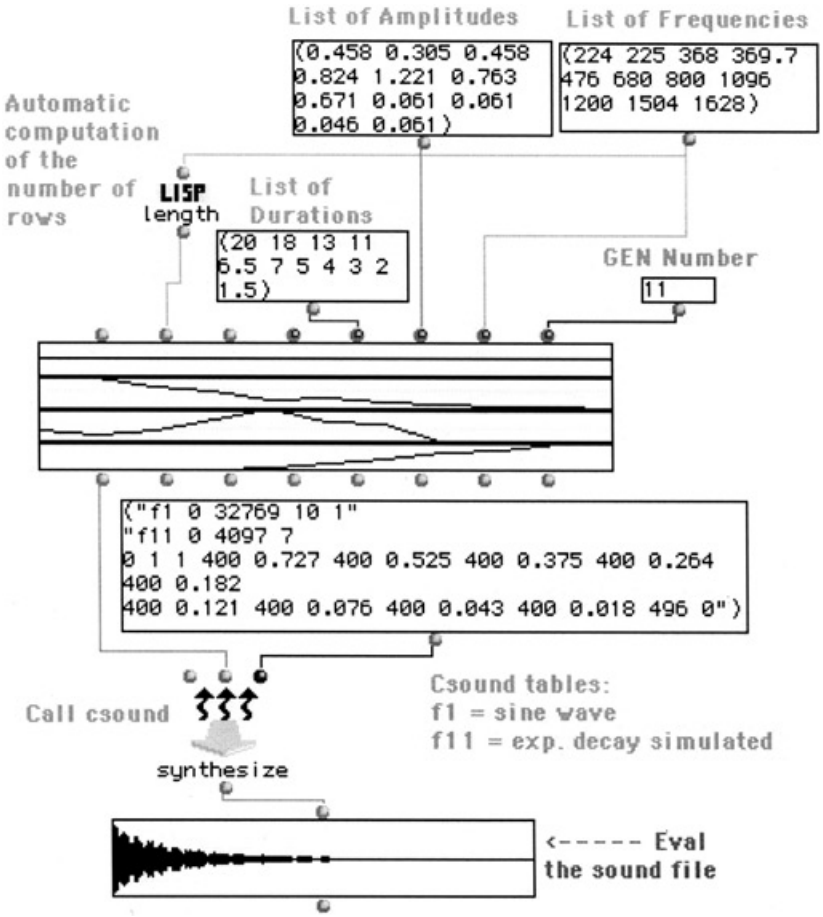


Figure 3. An omChroma patch defining a bell-like sound using Csound – from the omChroma tutorial.

sample rate and output file name. When this function is evaluated, the synthesis process begins and a sound file is generated.

3. A SIMPLE EXAMPLE

In this example a simple percussion instrument is made, consisting of a mass used to strike a tuned circular plate (figure 4). Since a force is applied to control the movement of the plate, a second plate (rectangular) is included, again linked to the mass by a strike connection, such that the mass is trapped between the two plates. This means that even if an upward (positive) force is applied for some time to the mass, it will always be kept within a limited distance from the intended strike target, and so only take a short time to impact when an appropriate downward force is used. Figure 4 shows the instrument construction. Here the force value (in Newtons) is set to 'dynamic' so that it can be controlled from OpenMusic functions. Figure 5 shows the items used to define an event class for this instrument.

In the figure, there is a ModInst factory containing the instrument. A text string is connected to the right input to name the instrument, while the left output is connected to get-modalyser-instrument in order to define the

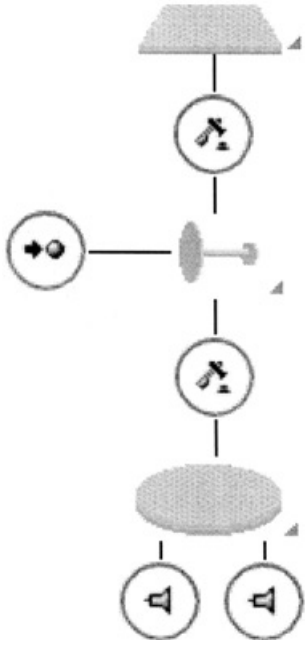


Figure 4. An MfOM struck plate instrument with a 'backstop'.

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

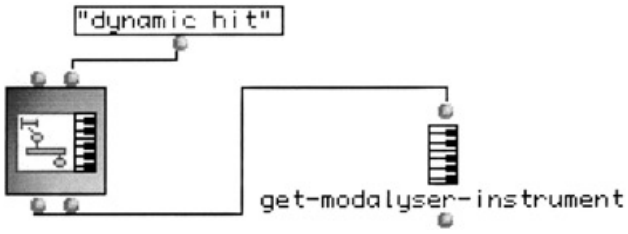


Figure 5. A ModInst factory connected to the get-modalyser-instrument function to define the event class.

event class for this instrument. After evaluating this function, the new event class can be used in a synthesis (and in fact the original ModInst is no longer required).

Figure 6 shows a patch using the new event class. At the top is a breakpoint function (bpf) factory that specifies the force values for the hammer. The *x* values of the bpf are treated as equal sub-divisions of the total duration specified in the synthesise function. This is connected to an event factory for our instrument, together with a number specifying the number of points in the control envelope. The left output of the event factory is then connected to the synthesise function, together with various parameters for setting up the synthesis (as labelled). Clicking on the left most input of the synthesise function gives a pop-up menu for selecting the synthesis engine to use – for our library it is *mfom*. The output of the synthesise function is connected to a *sound* factory. Evaluating this box will trigger evaluation of the entire chain, synthesising the sound and then loading the resulting sound file into the sound factory as shown.

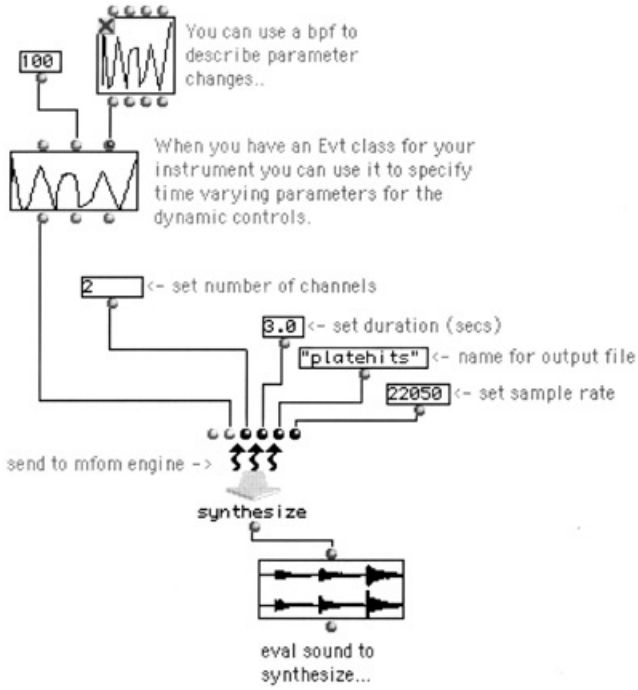


Figure 6. Breakpoint function control of a simple struck plate instrument.

4. MAPPING HIGH-LEVEL EVENTS

4.1. Mapping rhythm

This example uses a similar instrument to the one above, but now a mapping of high-level information onto the rhythmic control of the instrument is included (figure 7). A *voice* factory is added for our initial representation, giving us a simple rhythm in SWN. For convenience of calculation the voice is converted to a *chord-seq*, which provides simple access to the note onset times, durations and velocity values. It is a simple matter to map a MIDI file on to chord-seq, so this example can be easily modified for MIDI control. The chord-seq is passed to a subpatch (*hitevents*) that converts this information into a bpf specifying the hammer force over time. A simple subpatch is also used to calculate the duration of the synthesis from the maximum value in the onset list (which is the total duration) plus a half second to allow for some decay time. This duration is used by the synthesise function.

Figure 8 shows the contents of the hitevents subpatch. The mapping task is broken down into several key steps, each involving a loop sub-patch that carries out an iterative process. The loop *instxpoints* simply produces a list of numbers from 0 to (number-of-points – 1), number-of-points being the total number of values to use in the bpf – 100 in this example. These numbers are the *x* coordinates for the bpf. The loop *inststrikepoints* calculates the *x–y* coordinates for each note onset, using the onset times and MIDI velocity values from the chord-seq. The *x* coordinates are found by equating the ratios *x* : (number of points) and (onset time) : (total duration). The y coordinates are derived from linearly scaling MIDI velocity to negative force value, where the maximum velocity (127) is equivalent to –9.0 Newtons. The list output from this loop is sorted to ensure that the coordinates are in time order (which is not guaranteed prior to this point) and then passed to the loop *instypoints*. This loop uses the event coordinates to generate a sequence of number-of-points values for the hammer force. Since this instrument is un-damped, the durations of events are not used and an arbitrary time for a note's down-force is selected. From trial and error, two points at the down-force value is used here for a note trigger, but this value should be made dependent on the duration of the sequence and the number of points used in the control. The up-force used is +2.0 Newtons which is exerted at all times that an event is not being triggered.

While this mapping can be effective, the relatively primitive design has problems when interpreting dynamics, i.e. variations in MIDI velocity. First, it is not clear that a linear mapping of MIDI velocity to force is correct – since there is a non-linear relationship between force and velocity under Newton's Laws. An alternative would be to use a *speed* connection in the instrument (but this has other problems) or a more sophisticated calculation of the force values. Secondly, varying the

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

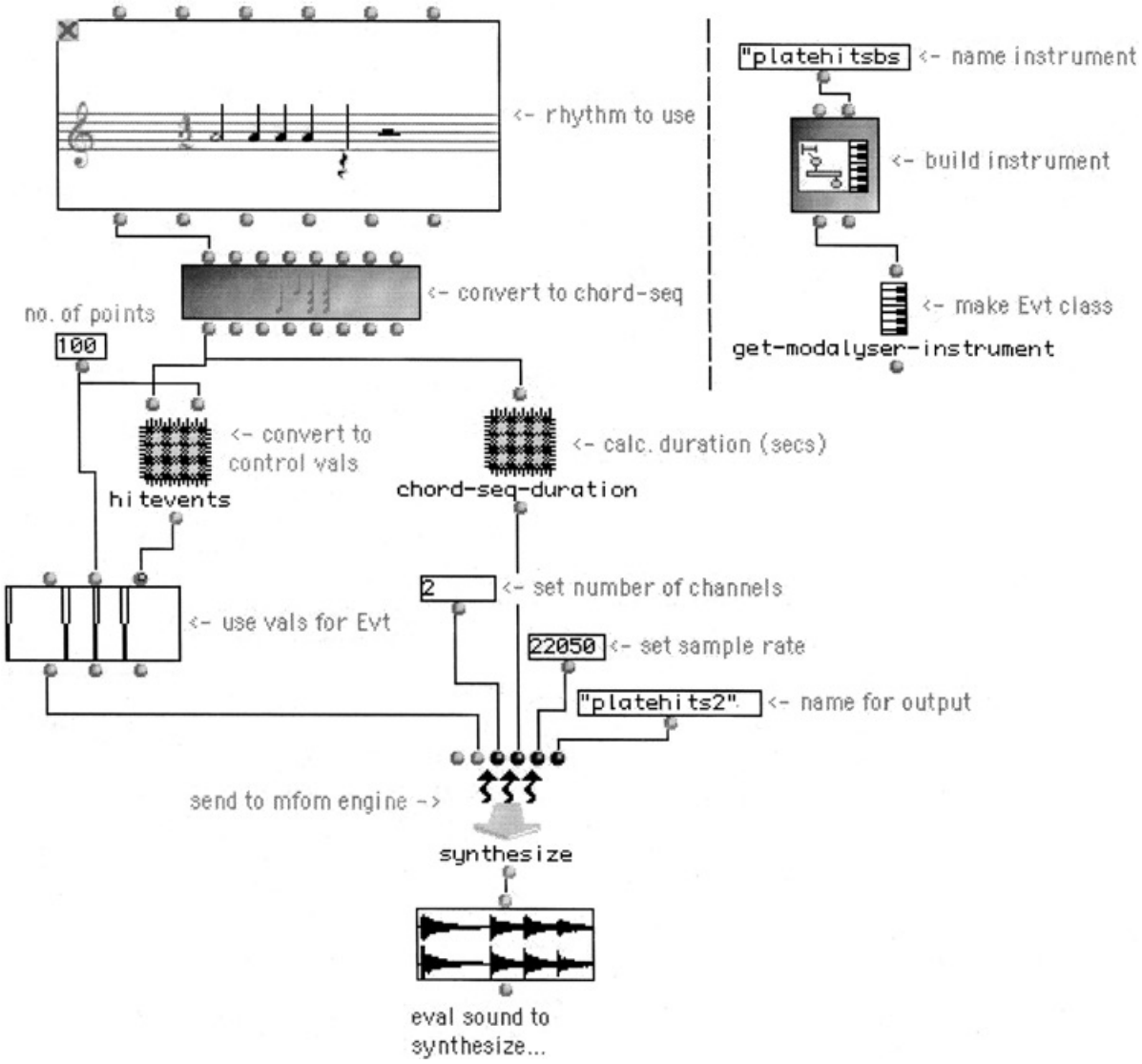


Figure 7. Control of a simple struck plate instrument with standard notation.

force necessarily changes the timing of the strikes, which is not taken into account in this example, thus low velocity (hence absolute force) events sound later than they should. While in human musical performance there may be a relationship between accent indications and variation of onset times from strictly as written, the variations here are too great. The solution would be to take the relationship of force to timing into account and so shift the down-force onset for softer notes to be earlier than for louder notes.

4.2. Mapping pitch and rhythm

While MfOM offers the user the ability to build whatever mapping algorithm they choose, using the visual programming capabilities of OpenMusic, it was felt that it would be useful to provide some initial mapping utilities. There are now a set of functions designed to map from OpenMusic's music classes (chord-seq, voice, etc.) to envelope shapes specifying different types of control values. There are two for producing pitch controls, a uni-dimensional one for finger on fingerboard positions and 'melting' between pitched objects, and a multi-dimensional function for opening and closing holes in a tube. There are also four excitation mapping functions for the following situations: plectrum position, bow speed (across string) and position (down onto string), reed breath pressure, and hammer (strike) force. A final function for use with these mappings is one which splits a polyphonic source into a set of monophonic chord-seq's, one for each pitch present in the original. This is useful for xylophone or piano-like instruments, where a separate input is required for controlling each pitched component. These functions are in their first implementation and are quite basic, providing a few controls over their behaviour and giving naïvely calculated outputs. One issue is the explicit separation of pitch control from other parameters for ease of reuse, while in actual

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

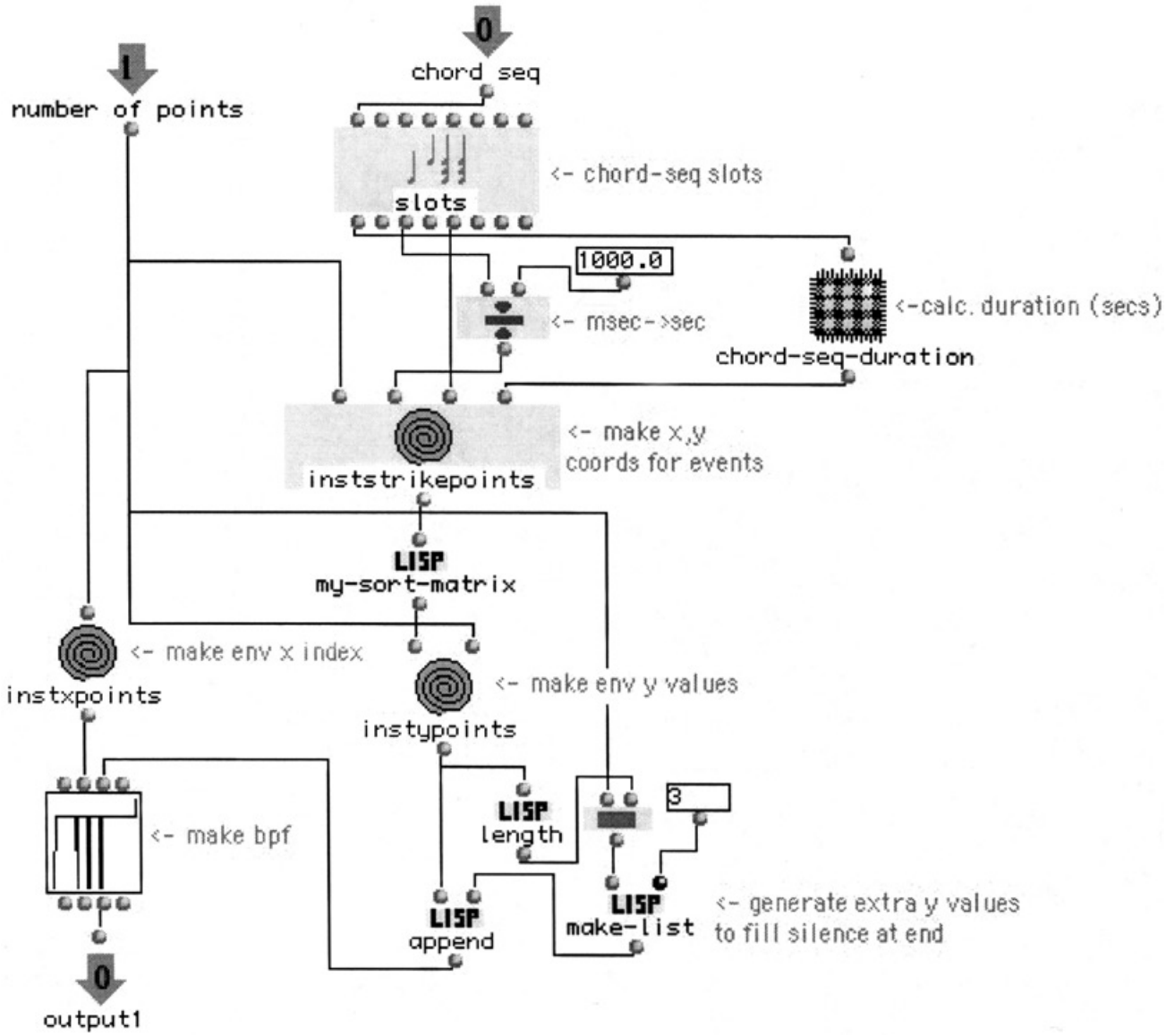


Figure 8. Mapping a chord-seq to a breakpoint function controlling force in a struck plate instrument.

fact there should be some interaction between these elements. For example, the breath pressure applied to a reed in a clarinet-like instrument should vary according to the pitch being played. However, it is believed these functions serve as a useful starting point.

A bowed string instrument is used in this example (figure 9), with a finger–fingerboard combination for controlling the instrument's pitch. Figure 10 shows two of the predefined mapping functions in use – *seq->bowpos&speed* and *seq->linepitch*. The first of these is given the music source (here a *voice*), a number of points to use in its output, a total duration time, the (vertical) position of the bowed object, the maximum down position of the bow (which must be lower than the bowed object position in order to press the bow against the bowed object) and the maximum (horizontal) bow speed. Both the bow position and peak bow speed will be varied according to MIDI note velocity from the input. In the figure, the output envelopes from this function can be seen in the event class. The central envelope gives the bow speed, which varies sinusoidally across the string, alternating direction between notes. The bottom envelope is the bow position, pushing down onto the string for each note. These shapes are a very primitive interpretation of the notation into bow action, with little input from actual performer behaviour, but can achieve reasonable results. The pitch control mapping is again given the music data, number of points and total duration, but is then given a base pitch, a maximum pitch (not used here, as this is for the situation where an object is morphed from one pitch to another) and a list of delta points. These govern the number of points over which pitch changes occur. Since it is a list, one can insert variable glissandi, which cannot be derived from

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

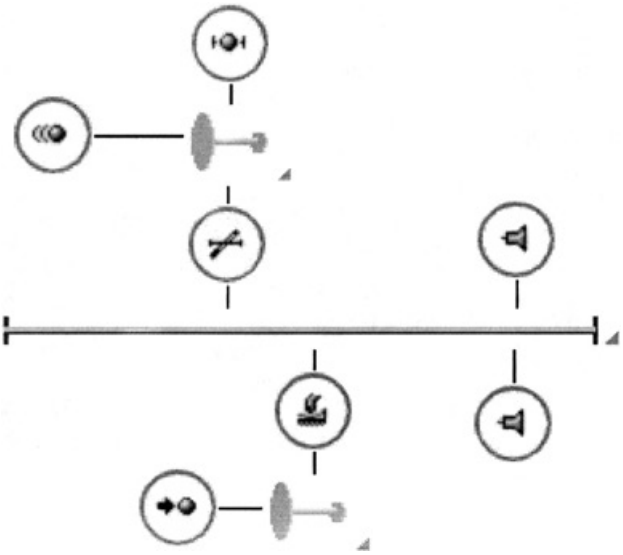


Figure 9. A bowed string instrument, using the top mass as the bow and the bottom mass as a finger with a fingerboard connection for pitching the string.

an input voice or chord-seq object. This pitch function then calculates the distance along the string, using the well-tempered rule that an increase in pitch of *n* semitones implies a frequency multiple of 2^(*n*/12), and the fact frequency is inversely proportional to length for a string – the base pitch provided must therefore be the frequency of the open string. In the figure, the finger position envelope is shown at the top of the event class.

A particular issue with a fingerboard system for pitching is that care is needed with the timing of the movement of the finger with respect to the movement of the bow, which can sometimes lead to loud artefacts (scrapes or knocks). Due to the normalisation of the output sound, these are problematic. In this case much 'tweaking' of the values used in the mapping functions may be necessary in order to gain adequate results. Again, a combined function mapping both pitch and excitation parameters may be able to automatically reduce these artefacts.

4.3. Parameterised instruments

While a gain in sophistication of control over Modalys-ER's system has already been shown, MfOM also provides tools for simplifying the construction of complicated instruments that are not present in Modalys-ER. This is done through a system of *parameterisation*. Having constructed a simple instrument, the user can pass lists of parameter values to get-modalyser-instrument which will make duplicates of the instrument inside the newly defined event class, each using a different parameter setting from the list. A common application of this is to automatically generate a polyphonic instrument from a monophonic template. The next example does exactly that, taking the struck plate instrument from earlier as a template. Figure 11 shows the patch for this synthesis. A MIDI file is used as the source data, and it is converted into a chord-seq for display purposes. (Note that the chord-seq object does not display the durations of notes, just pitch and onset timing. The notes in this example have various durations, but in the chord-seq they all appear as crotchets.)

From the chord-seq, a list of all of the pitches that are present is derived, duplicates are removed, the remainder sorted into ascending order and converted to frequencies. This task is carried out by the *unique-freqs* subpatch (detail not shown). These values are then used by get-modalyser-instrument so that our event class can play all of the notes present without the synthesis time overhead (and complexity of our Modalys-ER patch) that would arise from making the 127 duplicates required in order to cover all possible MIDI pitches. In order to specify how these values are to be used, the function is also given a list containing the object and parameter names to use – here 'New plate' refers to the plate to strike and *mo::pitch* sets the pitch property for the plate. There are utility functions in the MfOM library for discovering the available properties for different object types. Although here only the pitch for each duplicate is changed, it is possible to specify any number of parameters in this way (for example, randomly selecting the material for each plate as well as setting each pitch).

After evaluating the get-modalyser-instrument function, an event class is defined for this parameterised instrument. Notice (in figure 11) that this class has five control inputs, one for each hammer force, where each hammer strikes a differently pitched plate. The system prepends a number of asterisks to the name of each keyword input according to position in the list of parameters given to get-modalyser-instrument. Since the frequencies were sorted into ascending order, the lowest pitch hammer has no asterisks, the next pitch up one, the next two, etc.

In order to map the music data onto control information, the function *seq->strikeforce* is used, which, given a series of music events, will produce a control envelope suited to applying (positive and negative) forces to a hammer. Since there are several pitches in the source material, the function *seq->monseqs* is added to split the initial phrase into a set of chord-seqs, one for each pitch. This set is then passed as a list to the mapping function, which will output a control envelope for each one. The lisp functions first, second, third, and so on, then extract these envelopes and send them on to the event class. Clearly the ordering is important here, so that the correct notes arrive at the correct event input. This is ensured by the fact that both the unique-freqs subpatch and the seq->monoseqs function order by pitch from lowest to highest.

The parameter system is particularly useful since the

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

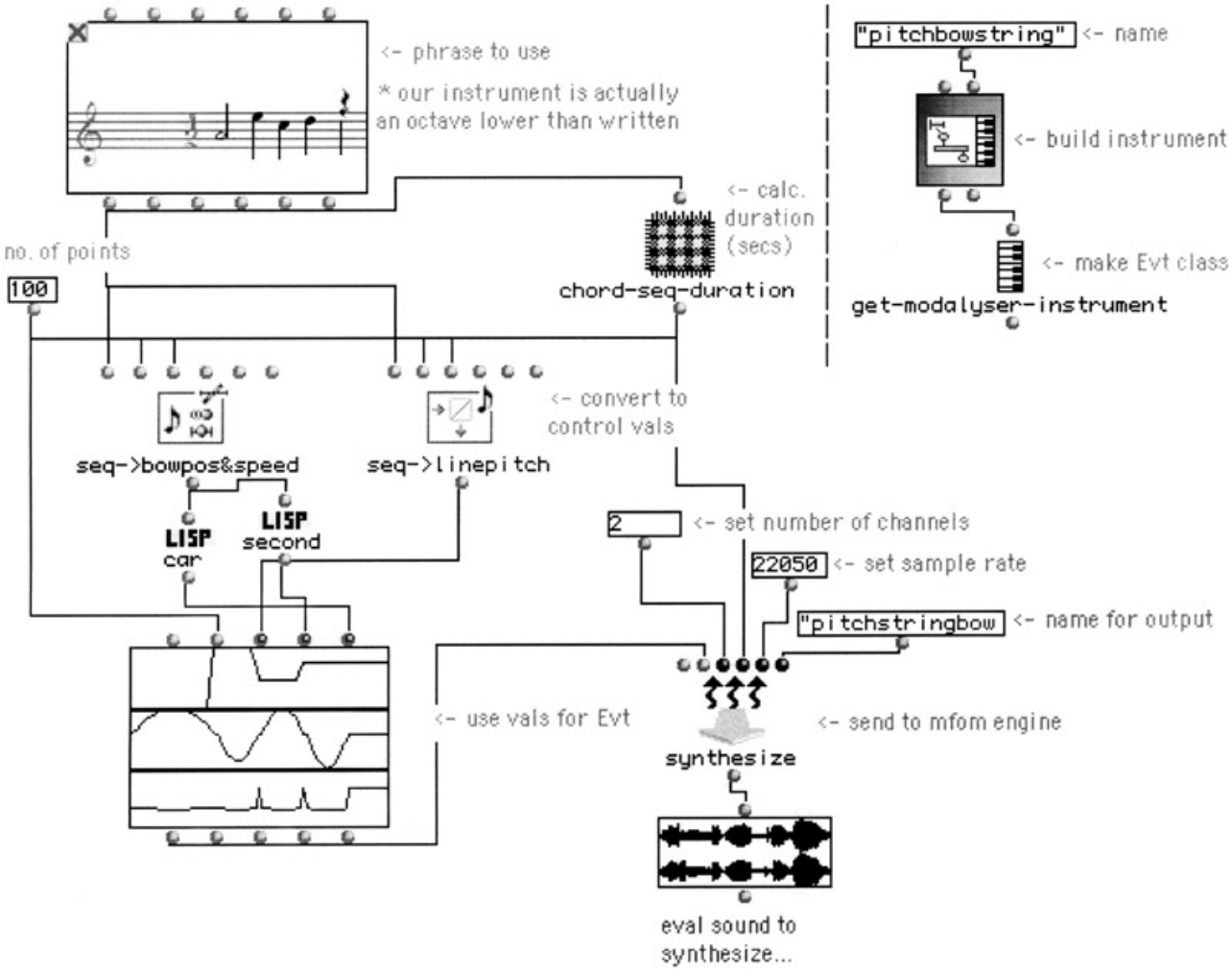


Figure 10. Control of a bowed string with variable pitch using standard notation.

process of creating large numbers of sets of objects in an instrument can be lengthy, particularly when changes need to be made to each set of objects after duplication. Using parameters, as many or as few duplicates as necessary can be created for a specific synthesis, the parameter values may be algorithmically derived in various ways (rather than by hand) and changes to the single original are passed on to all the copies. While the polyphonic application is simple for tuned percussion instruments, it would be more difficult to apply to a cello-like instrument. First, choices have to be made as to what pitch each duplicate should have (e.g. every interval of a fifth from the lowest note present) and in fact how many duplicates should be made. Secondly, once the instrument has been created, the control system must then select an appropriate string on which to play a given note – since in most cases the same pitch could be achieved on more than one of the strings (using different finger positions). Typically this last process is context dependent for a human performer, but in the synthetic case the limitations of finger spans and other aspects of human dexterity, which impact on the performer's choices, are not present. It is therefore not clear what rules one should apply in this case.

4.4. Automatic instrument construction

In addition to instrument parameterisation, the MfOM function *build-instrument* can derive instruments algorithmically and so again adapt instruments to the high-level events that they are designed to play. This function is similar to the 'Instrument Wizard' in Modalys-ER, and given a few parameters will assemble the instrument components accordingly. The function arguments are: resonator type (string, circular membrane, etc.), interaction type (e.g. strike, pluck, reed), list of pitches that the instrument should be able to play, material (to make the resonator from) and number of modes in the resonator (i.e. the number of vibration modes that will be used in calculating the motion of the object, with more modes giving higher quality results but extending calculation time). The detail of the instrument (a ModInst instance)

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

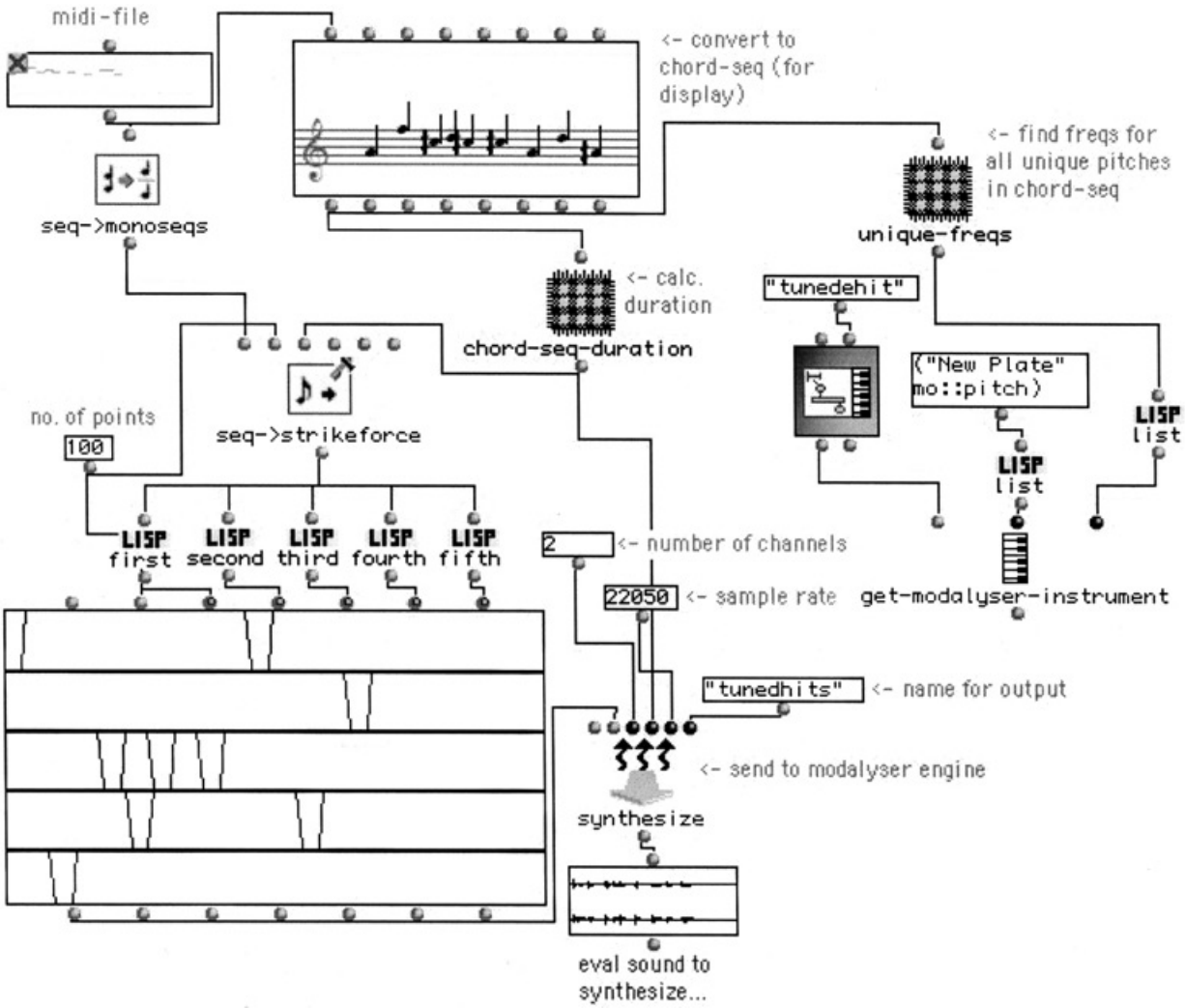


Figure 11. Automatic parameterisation of pitch in a struck plate instrument.

produced will depend mainly upon the first three parameters, while the last two specify single parameters. For example, if a tube resonator is used with several pitches, a series of dynamically resizable holes will be positioned appropriately along its length. If a strike interaction is specified, a mass will be created to strike the resonator, together with a dynamically controlled force for moving it and a 'backstop' to constrain its distance from the resonator. Figure 12 shows a patch using build-instrument, while figure 13 shows the bowed string instrument created. The first two inputs are set to string and bow, but are not visible (they have been selected from pop-up menus). The unique-freq subpatch is again used to find all the required frequencies, which is then used as an input to build-instrument and as before to the seq->linepitch function. In this case, build-instrument will ignore all but the lowest frequency and use that to set the pitch of the string. A more sophisticated solution should examine the highest frequency also and use that to ensure that the bow position on the string is kept further along the string than the finger position required to reach this pitch. The constructed instrument is passed to get-modalyser-instrument, and the control side of the patch is much as before. Again, a key advantage of using the build-instrument is that the instrument will be automatically adjusted to deal with the music information it is being required to play. While here, setting the pitch is a relatively simple task; for more complex cases such as tubes that use holes for tuning and pitching, this system can greatly enhance usability.

5. OM_MODALYS

As mentioned previously, there is another OpenMusic library for use with modal synthesis – OM_Modalys (Lanza 2002). In its latest form (version 2.0 for October 2002 release) it provides a close mapping of Modalys' Scheme programming interface into OpenMusic (since Scheme is a Lisp-like language, this almost direct correlation is practical). This has some advantages, particularly for current Modalys users, but also means that OM_Modalys patches are generally

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

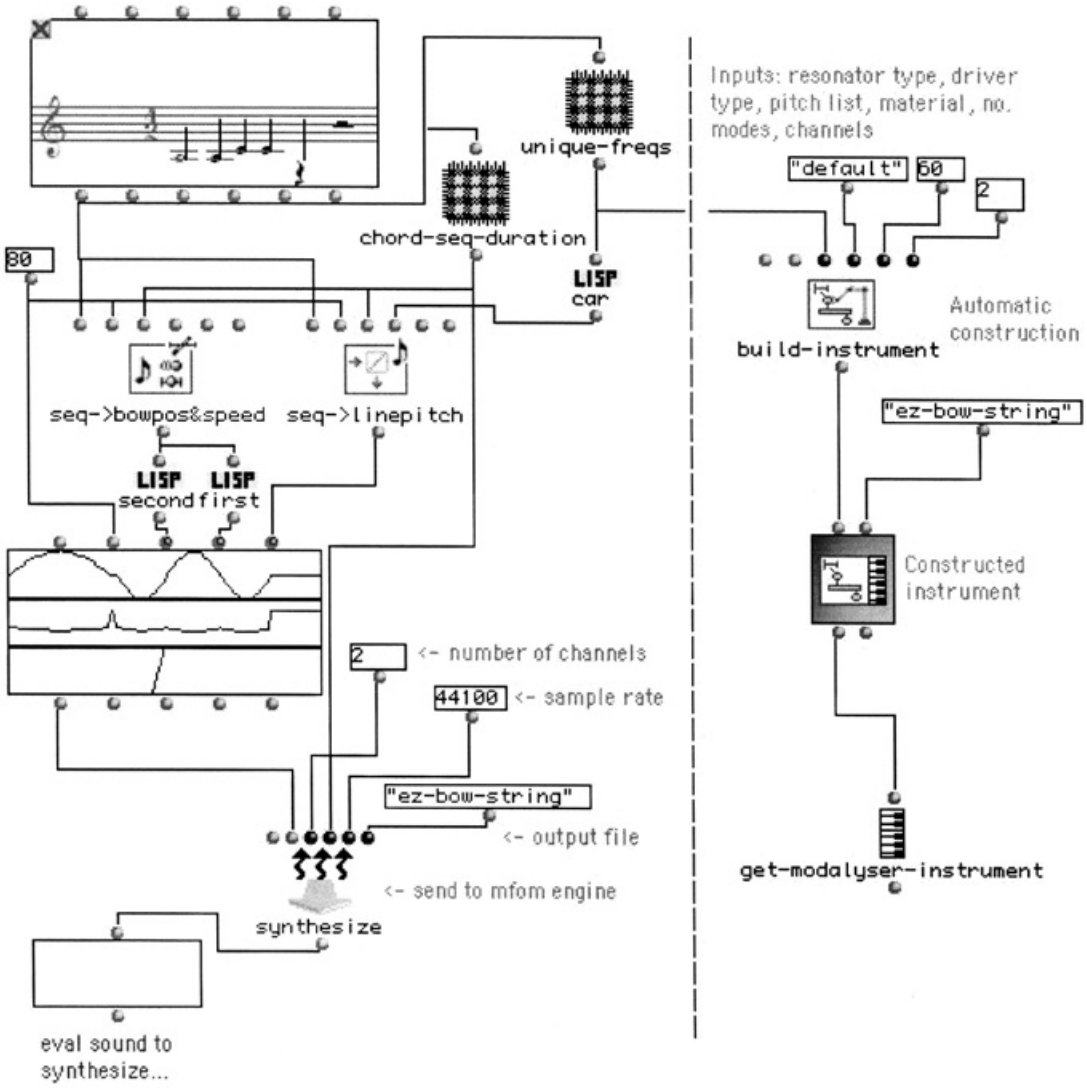


Figure 12. Using the build-instrument function.

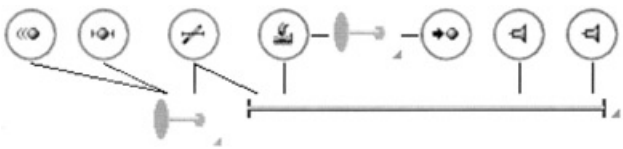


Figure 13. A bowed string instrument (with fingerboard) produced by build-instrument: (from left to right) speed and position connections to a mass (the bow); a bow connection to the string; a fingerboard connection and mass (acting as a finger), with a force connection (representing finger pressure on the fingerboard); two pickup connections (to convert the string vibrations to sound).

much more complex and lack MfOM's more structured approach and enforced syntactic correctness. This last point is to say that MfOM attempts to guarantee that a meaningful Modalys program is produced and, for example, limits parameter values to valid ranges. Also, the OM_Modalys library requires the Modalys application for synthesis (much as the original Modalyser did) since its output consists of text files containing Modalys code. Further work on the communication between the internal Modalys synthesis engine and OpenMusic should allow the OM_Modalys library to internally synthesise sounds also – the limiting issue currently being error handling. This library is seen as complementary to MfOM, providing another level of synthesis specification for more technically advanced users. As such, a translation function which allows automatic conversion of an MfOM instrument to an OM_Modalys patch has been developed. Figure 14 shows a patch illustrating the different levels at which a Modalys synthesis can be now specified, while figure 15 shows a part of the instrument converted to OM_Modalys (the patch is too large to show fully). It is now possible to go all the way from build-instrument, requiring only a few parameters, to ModInst and its graphical editor, through to OM Modalys and its detailed representation, and finally back to Modalys' Scheme programming.

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

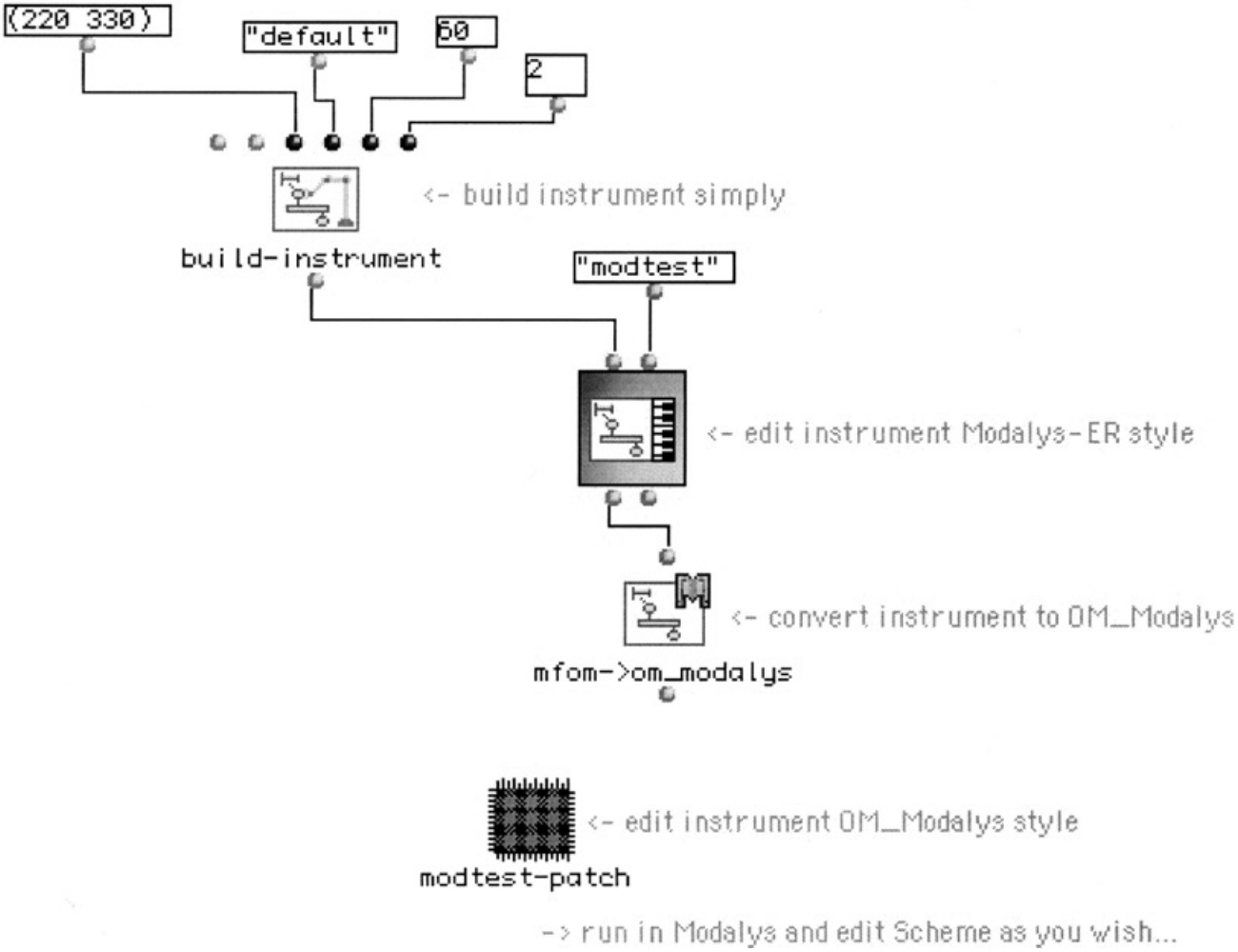


Figure 14. Instrument definition at multiple levels: build-instrument, ModInst, OM_Modalys patch, Modalys Scheme.

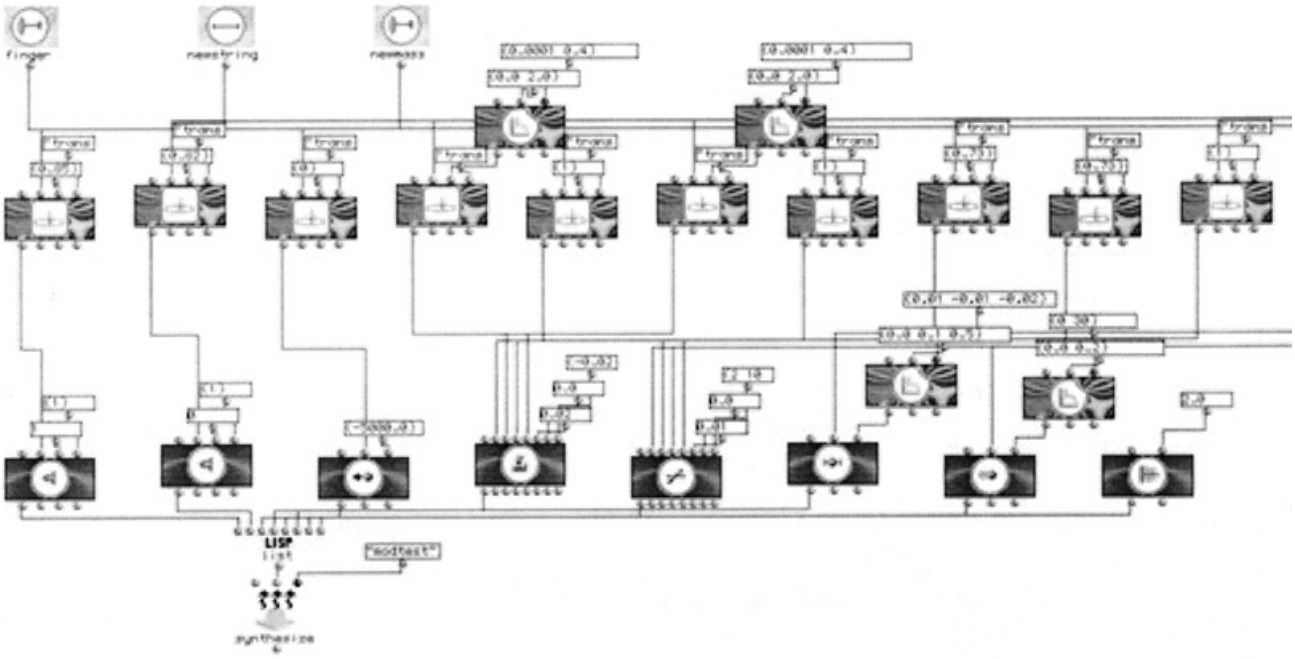


Figure 15. Part of a ModInst instance converted to an OM_Modalys patch.

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press

6. DISCUSSION AND FURTHER RESEARCH

OpenMusic with the MfOM library provides an integrated system for physical modelling sound synthesis with sophisticated facilities for developing performance control. A synthesis is carried out via a two-stage process: an instrument design stage followed by performance specification. This provides a clear demarcation of roles and fits well with the existing omChroma library for generic synthesis specification within OpenMusic. However, the aspect of synthesis control remains a complex problem. While the construction of a virtual instrument is relatively simple, where users can apply their existing knowledge and experience of acoustic musical instruments, direct specification of physical interaction parameters, such as forces and finger positions, is an unfamiliar experience for most composers. In these examples, some of the potential for constructing mapping systems from typical high-level musical representations down to the physical control level has been illustrated. The design of such mappings requires a reasonable level of Lisp programming knowledge, and so some initial mapping functions have been provided that can, with some patience, produce reasonable results. Given that these functions are relatively unsophisticated, the development of truly effective systems remains a challenge. This is particularly so when taking into account aspects of musical interpretation. That is, not just a literal mapping of notation as written, but inclusion of the subtle variations in timing, dynamic and timbre imposed by human performers when playing according to a particular genre. Much research in this field has already been undertaken (e.g. Desain and Honing 1997) and results from such work may provide valuable data for application here. Physical modelling synthesis controlled from a Lisp-based algorithmic environment has also been carried out by Laurson *et al*. with ENP, PWSynth and the Patchwork environment (e.g. Laurson and Kuuskankare 2001). However, their work has concentrated on precise simulation of traditional acoustic instruments (using waveguide models) and their associated score interpretation. ENP provides standard Western notation with useful extensions that are not available within OpenMusic. For example, *score bpf*s, which provide break point functions aligned to a particular stretch of score. Such an extended notation system would be useful for future development of this work, but the problem of handling notation mapping for arbitrary user-defined instruments will remain. The constraints systems used by Laurson (*performance rules*) may be usefully adapted for application here, but this has yet to be investigated.

More desirable than the basic mapping functions described in this article (which only cover a limited range of control situations) would be an intelligent system that, given an instrument and some form of score, would automatically derive an effective mapping between the two (where possible) – a true 'virtual musician'. While in the immediate term the aim is to continue development of the set of mapping functions to accompany the library, over the longer term it is hoped to pursue research into intelligent systems. OpenMusic should provide an effective tool for this as Lisp has as a common platform for AI applications. Given that Modalys-ER and OpenMusic are developed using the same programming tools, it should be possible to port any such system back into Modalys-ER, so that control of synthesis from SWN or MIDI could be provided within a dedicated synthesis environment that does not require the level of programming knowledge associated with OpenMusic.

REFERENCES

Agon, C., Stroppa, M., and Assayag, G. 2000. High level musical control of sound synthesis in OpenMusic. *Proc. of the ICMC Berlin 2000*. ICMA.

Assayag, G., Rueda, C., Laurson, M., Agon, C., and Delerue, O. 1999. Computer assisted composition at Ircam: PatchWork & OpenMusic. *Computer Music Journal* **23**(3): 59–72. MIT Press.

Desain, P., and Honing, H. 1997. Structural Expression Component Theory (SECT), and a method for decomposing expression in music performance. In *Proc. of the Society for Music Perception and Cognition Conference* **38**. MIT Press.

Johnson, P. 1992. *Human Computer Interaction: Psychology, Task Analysis and Software Engineering*. McGraw-Hill.

Laurson, M., and Kuuskankare, M. 2001. PWSynth: a Lisp-based bridge between computer assisted composition and sound synthesis. In *Proc. of the ICMC Havana 2001*. ICMA.

Morrison, J. D., and Adrien, J.-M. 1993. MOSAIC: a framework for modal synthesis. *Computer Music Journal* **17**(1): 45–56. MIT Press.

Polfreman, R. 1999. A task analysis of music composition and its application to the development of Modalyser. *Organised Sound* **4**(1): 31–43. CUP.

Polfreman, R., and Loomes, M. J. 2001. A TKS framework for understanding music composition processes and its application in interactive system design. In *Proc. of the AISB'01 Symp. on Artificial Intelligence and Creativity in Arts and Science*, pp 75–83. SSAISB, York.

Vercoe, B. 1993. *Csound: A Manual for the Audio Processing System and Supporting Programs with Tutorials*. MIT Press.

https://doi.org/10.1017/S1355771802003126 Published online by Cambridge University Press