THE PHYSICS NETWORKS SCRIPTING LANGUAGE TO DESIGN MASSES-INTERACTIONS PHYSICAL MODELS FOR ARTS

Nicolas Castagné
Univ. Grenoble Alpes, Grenoble INP, ICA,
38000 Grenoble, France
Nicolas.castagne@grenoble-inp.fr

Annie Luciani
ACROE
38000 Grenoble, France
annie.luciani@acroe-ica.org

ABSTRACT

Masses-interactions modeling, also called physics Newtonian networks modeling, was pioneered in the late 70's with the CORDIS-ANIMA formalism. It is a modular physics-based approach, used in sound design, musical instruments design and musical composition, and also in computer graphics animation, virtual reality, gesture interaction, and more generally for digital creativity.

Within the research that progressively position physics-based modeling as a core mean to improve creativity with the computer, the paper introduces the Physics Network Scripting Language. PNSL is aimed at accompanying the modeling processes of any user, including artists, with masses-interactions modeling. It serves in particular, but not only, sound and musical purposes.

The paper describes the PNSL language: context, goals and general choices; labeling system tailored to support modeling complexity by managing freely large module sets; main features and syntax choices; and implementation and exemplary usage contexts.

1. INTRODUCTION

Current research in the field of physics-based modeling for sound and music question various axes, including:

1. Research toward innovative modeling tools, able to serve the practice of physics-based modeling within creative processes – that is: enabling the user, *e.g.* a musician, to design his/her own physical models.

2. Research on core physics-based computational techniques and algorithms, such as works aimed at improving the precision of digital solving of partial differential equations, *e.g.* in non-linear problems such as [1], or in collision such as [2], etc.

3. Research toward new models. This includes high fidelity modeling of existing acoustical instruments and structures, but also adaptable generative models aimed at sound/musical quality, but without necessarily referring to a real instrument. Today's research focus in particular on non-linear phenomena, such as non-linear resonators, on multisensory models (sound + visuals + gesture), etc.

4. Research on musical use and musical relevance of physical modeling, especially regarding incorporation of modeling at the core of creative processes.

This paper is a contribution to the 1st of the above-proposed axis, hence may serve developments along the 3rd and 4th axis.

There is today an increasing use of predesigned physics-based algorithms within digital musical instruments, which probably relates to the sound quality enabled by simulation. Beyond, work along the 4th research axis above progressively demonstrates that the design of physical models by musicians themselves has musical relevance – see for example [3] and [4]. However, and though such promising research results, the practice of physical modeling itself is still rare among musicians. For example, it is still far less incorporated in daily creative practices than signal-based patching.

As a hypothesis for this fact, one could remind that designing a physical model can be particularly difficult. We assume that another reason lies in limits of available modeling/creative tools, besides past successful achievements. For example, we assume that the physical modeling activity hardly well incorporates into today's usual signal-based musical tools. Given the hope to progressively turn physical modeling into a daily creative practice, there is still a need for innovative modeling and creative technologies: adequate modeling tools, modeling languages, modeling GUIs principles, etc. This is today a challenge for the research in software design in our fields.

As a contribution to this context, this article introduces the Physics Network Scripting Language, PNSL.

Regarding the core modeling/simulation approach, PNSL roots on the masses-interactions networks physics-based approach, as defined in the CORDIS-ANIMA formalism. It was pioneered in the late 70's by Claude Cadoz during his PhD thesis, then first published in the 80's, *e.g.* in [5]. Within this modular formalism, a model consists in a network of masses bilaterally interconnected with physical interactions. Similarly to Kirchhoff's networks developed in electrical engineering, the approach fits within lumped modeling systems. Similarly to [6], it is a type of block-based representations connected by bilateral links. In this context, CORDIS-ANIMA networks are particularly adapted to target any dimensionality (scalar, 1D, 2D, 3D), for uses in Computer Music, Physics, Computer Animation and Haptics. Given these various properties and uses, CORDIS-ANIMA networks have been called in [7] "*Physics Newtonian Networks*", or shortly "*Physics Networks*". [8] discusses the approach with others in the field.

The approach has so far supported the design of tens to hundreds of thousands physical models. It has been employed in about one hundred artworks – counting only those with contribution from ACROE. Regarding only models aimed at sound and music, examples include [9]: linear and non-linear vibrating sound structures (strings, acoustic tubes, membranes, plates, bars, gongs, etc.), sound boards and resonators of many sorts, instantaneous and continuous excitation means (striking, plucking, bowing, blowing, scrubbing, etc.), modifications of the vibrating structure (damping, pinching, pulling, deforming, fracturing…), sound propagation models (room effects, models aimed at sound spatialization in multi-channel audio, etc.), models of sound transformations and processing, etc. Beyond sound synthesis and instrumental models, works target also generation of musical events, and musical composition. As to them, the approach supported the invention of quite new compositional processes, centered on the physical modeling activity itself, such as Cadoz' "compose (with) physical modeling", first discussed in [3]. Finally, besides musical purposes, the approach developed also in Computer Animation and Virtual Reality [10], gesture interaction including force-feedback [5], and more generally as a mean to generate dynamic phenomena aimed at the human sensori-motor skills and at creativity with the computer.

On the side of creative tools that support CORDIS-ANIMA masses-interactions modeling, the GENESIS series [11] is a series of modelling/simulation GUIs aimed at instrument and sound design, and at musical composition by means of physical modeling. Other related technologies include for example synth-a-modeler [12], Cymatic [13], the PMPD extension for PureData [14]. Also, the masses-interactions approach progressively spreads within other audio/musical software, such as Max/SP, PureData, Processing and others. However, along with these existing technologies, there still is a need for a generic modelling language for creativity with the approach.

The paper presents a domain-specific language [15], called the *Physics Network Scripting Language* (PNSL), that has been designed by considering the properties of masses-interactions modeling and tailored to support the modeling processes of the user, including artists, toward any model falling into the scope of masses-interactions models, especially in creative contexts.

Rather than a detailed description of the language, that could not take place in a paper, we first review our objectives with PNSL and then we introduce its main structure, characteristics and implementation. This includes a discussion of the labeling system, tailored to support managing each single module, as well as very large module sets, when building complex networks.

2. PNSL - GOALS AND SPECTRUM

We aimed at supporting with PNSL the design of any model falling into the spectrum of masses-interactions modeling, as formalized in CORDIS-ANIMA [5].

2.1 PNSL, a language to practice CORDIS-ANIMA

A CORDIS-ANIMA model consists in a network composed of interconnected elementary physics-based modules taken from two categories: material points (called <MAT>), and interactions between two material points (called <LIA>, for the French *LIAison*). <MAT> modules support inertial behaviors, and <LIA> express physical influence between <MAT>. Connection between <MAT> and <LIA> is always bidirectional: they exchange with each other's a bidirectional data flow of dual variables, forces and positions. Hence, a <MAT> receives forces (intensive variables) from the connected <LIA>, and outputs to them its position (extensive variable), whereas a <LIA> outputs two opposite forces computed from the positions received from its two connected <MAT>.

The exchanged variables, forces and positions, may be 1D, 2D or 3D, depending on the dimensionality of the targeted phenomena.

The <MAT> and <LIA> categories further derive in a series of basic "module types". Each module type models an elementary physical behavioral law, through an optimized algorithm, with specific physical parameters. A <MAT> can be, for example, a simple mass, a fixed point, a viscous mass, etc. And a <LIA> can be many types of interactions, either linear or non-linear: potential such as attraction, elastic, cohesive, plastic with hysteresis, or dissipative such as viscosity or dry friction, etc. With such variety of <LIA>, it is possible to model percussions and collisions, including prolonged contacts, cohesions, repulsions, bowing or plucking interactions, irreversible breaking, etc. Non-linear capabilities, when needed, are included within the <LIA> modules themselves, under the form of non-linear profiles functions or finite state automata - see *e.g.* [5, 10, 11].

The number of elementary module types is small - noticeably smaller for example than the core library of modules in most signal-based patching systems. Each algorithm is rather simple, though physically meaningful. Hence, complex behaviors emerge from the building of large models made of many elementary modules interconnected into a network, through a constructive modeling process.

PNSL supports all module types defined in CORDIS-ANIMA, no matter of the employed simulations frequencies, and dimensionalities.

2.2 PNSL, a modeling language for varied uses

PNSL is centered on the modeling activity with CORDIS-ANIMA. It was aimed at supporting the user progressive process toward a model, rather than as a tool to provide a static description of a given model. It seeks providing the necessary but sufficient means to cope with the building of any masses-interactions network with all its properties, from the simplest to the most complex, no matter the foreseen usage.

The physical phenomena simulated by a physics network can be heard through a loudspeaker (provided it generates centered oscillations in the appropriate frequency ranges), but can equivalently be visualized on

screen, or *e.g.* used to generate the feedback of a force feedback device. Hence, PNSL was made neutral in regards to the aim pursued with modeling. It qualifies for the design of both mechanical and acoustical models, in either Computer Music, Computer Graphics, Virtual Reality, Haptics or multisensory real time synchronous simulation.

In regards to its potential musical uses, PNSL was also designed to be as neutral as possible. Hence, though it can serve in particular musical creation, it does not embed musical vocabulary nor musical notation of any sort. Similarly, though physical modeling especially enables modeling the instrumental universe, the language itself does not provide predesigned instrumental blocks, such as e.g. membranes or air tubes. It is conversely a mean to build, and to play around as wished, such instrumental, sound, or musical elements. In this way, it is a tool that hopefully lets musicians freely explore and write the still-largely blank page of musical creation by means of physical modeling.

This being said, and though this article focuses on PNSL basics, a library of scripts is progressively built and shared among users, with various routines that correspond to specific sound and musical model parts or goals.

2.3 Other General Goals with PNSL

Targeting the end user. PNSL users can be for example artists, composers, animation designers, etc. who may not be used to Physics nor to programming. Consequently, we sought a low entrance cost of the language, and its learning curve was an important factor. However, some users are skilled programmers, and the language hopefully allows them to benefit from their higher programming skills.

Adaptability. The features of the language can be adapted to the context of the target application at hand. In particular, the available module types, and, following, the categories of physics networks the user can define, can be varied from a target application to another. Also, in a given application, it is also possible to restrict the set of available features to those needed in the current modeling phase. For example: let the user access only the features related to the topology of the network, or only those related to physical parameters editing, or those related to initial state of the modules, etc.

Efficiency. Search for generality required possibility to manage large networks, possibly with hundreds of thousands <MAT> and <LIA> modules.

Expression power. Typical models' sizes and structures impact also the needed expression power. In particular, an analysis of past existing models and their design process showed that the modeling user needs handling large overlapping sets of modules at a glance, but also requires being able at any time to select and target any single module anywhere in the network. This required designing features able to target modules either individually or globally. It was achieved through the *labeling system*, which is presented in section 4. It also implied that any PNSL command should be able to process numerous attributes in various modules, with a single call and hopefully a concise syntax.

3. PNSL - GENERAL DESIGN CHOICES

This section situates our design choices regarding PNSL in the field of Domain Specific Languages [15].

3.1 PNSL: a Scripting Language

There is no agreement in the literature of Domain Specific Languages regarding whether or not it should be an *executable* language [16]. Though, PNSL aims at accompanying the modeling user throughout his/her modeling activity toward the final model, including exploration and trial and error loops. We hence stated that PNSL should *not* be descriptive or declarative languages only able to describing the state of a given model. Conversely, PNSL is a full-featured programming language, and enables performing any modification of the model, or browsing any of its properties, at any time. We further chose to set up a script language, as opposed to compiled languages, so as to increase the ease of use (*e.g.* easily launch a single command) and embedding possibilities.

3.2 System Front End pattern

PNSL follows the *System Front End pattern* of Domain Specific Languages, as defined in [17]: the end-user language itself is offered as a programmable front end, called PNSL_front, that allows accessing an underlying core data structure and application programming interface, called PN_Core.

Though, the choice of the System Front End pattern was not carried out to ease the management of a preexisting system, nor to reduce the development effort of PNSL, as considered in [17]: both PN_Core and PNSL_Front were developed altogether. We rather adopted this pattern as a mean to enable cohabitation of multiple front-ends on top of the core data structure of the model being designed. Indeed, in our case, the System Front End pattern made it possible to articulate the language itself (PNSL_front) with WIMP and direct manipulation featured in interactive graphical user interfaces. Hence, the new releases of GENESIS [11] and MIMESIS [10] now offer graphic/textual cooperation, and become multimodal modeling frameworks - in the sense of *representational* or *interaction* modality, as defined in HCI.

3.3 Language Extension and Specialization patterns

PNSL employs the Language Extension pattern [17] – or Embedding Pattern [18], or Extension Approach [16] – and the Language Specialization pattern [17] in order to benefit from and adapt the syntax, semantics and libraries of an existing full-featured language.

We chose in our case the *Tool Command Language* Tcl [19], by considering the simplicity of Tcl's grammar; its procedural, non-object oriented style; its promotion of variadic commands; its generalized use of strings, which is quite adequate to the PNSL's labeling system presented in the next paragraph (though using strings as the fundamental data representation also has drawbacks); and the fact that mathematical computation is rarely an important need in the context of masses-interactions

CORDIS-ANIMA modeling processes, so that it is worth using Tcl's `expr` command. Though, to further accommodate users, other front ends may be deployed for PNSL in the future within another language, with a reasonable development effort thanks to the use of the Front End pattern.

4. LABELING SYSTEM

A masses-interactions CORDIS-ANIMA model may exhibit a large number of modules, possibly millions. Acquired experience shows that real-cases models gather both *finely structured* network parts (where modules are interconnected one-by-one into a specific topology), and *regular network* parts (with some regularities in the network's topology). Similarly, a model often has parts with homogeneous attributes (physical parameter, initial states…), but also parts with non-homogeneous attributes, where for example each single module is given carefully chosen parameter values.

Given the above, the modeling process often requires handling various sets of modules together, and importantly these sets often overlap with each other's. But conversely the modeling process also requires being able to target any specific module separately, in particular, but not only, when it comes to assemble the modules point-to-point, or to deal with initial state of the modules.

Hence, the means provided to the user to reference both *any individual modules* and *numerous overlapping sets of modules* were paid specific attention. Additionally, these means had to be manageable both through the language, and through the GUI. The Labeling System was specifically designed by considering these aims.

4.1 A review of known concepts to cope with complexity in modular frameworks

In modular modeling frameworks, and in particular in creative contexts, when facing the need to let the user structure large datasets, most systems build on the principles of *encapsulation*: "capsules" or "macro modules" are created by the user by gathering various sub-parts – or, recursively, inner capsules – into a super structure. For example, encapsulation is a popular mean to cope with complexity in the context of signal-based modular patching environments such as PureData and MaxMSP, but also Simulink and others in other application fields. Similarly, in 3D shapes modeling, encapsulation is usually tied to the notion of object: various "objects", that may be rather complex, such as a sphere, a house, a character, etc., are placed into the scene. Tied to the notion of objects, 3D modeling also often promotes a hierarchical structure for the scene, through the notion of scene graph: an object can be made of sub-objects, in a recursive manner (e.g. a house made of walls, made of bricks, etc.). Noticeably, encapsulation also powers a number approaches to physics-based modeling, such as for example the BlockCompiler system [6].

Indeed, encapsulation can be seen as a virtue of Informatics. It enables drawing boundaries around the "capsules", hiding their inner complexity, and minimizing the "surface" or "outside" of these capsules. It enables coping with complexity by promoting a bottom-up hierarchical approach to complex modeling activities. And, importantly, it promotes reusability.

However, *encapsulation is not appropriate to masses-interactions networks modeling*. Indeed, complex physics networks do not match a strict tree-like structural organization. This relates first to the fact that, by principle, masses-interactions networks are coupled structures, where any part of a model is *interacting with* (and not *acting upon)* the other parts. It also relates to the idea that, with PNSL, we wanted the user to gain access to the elementary level at each and any corner of modeling process, so that *e.g.* musicians could experiment with "composing the inner of the sound" (as J-C. Risset founded), rather than managing only macro categories of vibrating objects. Indeed, past experiments showed that featuring the principles of encapsulation in PNSL would require the user to access individually, quite often, the inner modules of a capsule, for example to change series of parameters, or to modify a <LIA> connection inside the capsule or with another module in another capsule, etc. Symmetrically, it would require letting each "capsule" exhibit most, possibly all, its inner properties on its "surface", including its inner topology (e.g.: letting all the <MAT> of a capsule accessible to outside connections) and the its modules' physical parameters. This contradicts the core interests of encapsulation.

Beyond encapsulation, the labeling system is our proposal to cover the need of dedicated features to cope with complexity while modeling with masses-interactions Physics Networks.

4.2 Labels and Containers

In PNSL, a label is a string that refers to a module. We say that a label *targets* a module.

Upon creation, a label is said to be *declared*; upon deletion, it is *forgotten*. Declaring and forgetting labels never creates nor deletes modules themselves. It only deals with modules' labels.

Each module is given by the system a unique *permanent label* upon instantiation, in the form `@<integer>` (e.g. `@147`).

Then, the user can define for each module as many *user labels* as needed, that is: as many labels as there are contexts in which he/she wants to name, select, connect, parameterize, or more generally handle the module. *User-defined labels* can feature any number of slash (/) separators. Hence, `/string67/extermities/lia1` could be a label. Based on such separators, user labels are stored as a tree structure. A walk through the tree from the root "/" down to a leaf corresponds to a label, and refers to a module.

The sub-tree below an internal node, or symmetrically the labels sharing a part of their path from the label's tree root, can be seen as a context in which the user needed to name, or more generally consider, the targeted modules altogether. Each substring between two slashes separators in a label is hence called a *contextual name*.

A	`@23`	Picks a single module, through its permanent label.
	/oscilator1/stiff_lia	Picks a single label.
B	`@*`	Picks all the modules' permanent labels in the model (or, simply "all the modules").
	/strings/>>	picks all the labels starting with `/strings`, down to the leaves in the label tree.
	/membrane/>	Picks all the labels in the container `/membrane`, without descending further the tree.
	/audio_outputs/mass+	picks all the labels that start with `/audio_outputs/mass`, followed by an integer.
	`@12..87`	picks all the modules' permanent labels between `@12` to `@87`
	/strings1/#5..8	Picks the labels for the 5th to 8th based on the contextual names' indexing
	/rhythm_gen/RG1/mass11..	In `/rhythm_gen/RG,` pick modules named with `mass` followed by an integer >= 11.
C	`LIA:@*`	Picks all the <LIA> (i.e. all the interaction modules) in the model.
	`3D:@SELECTION/*`	Picks all the module with a `3D` dimensionality that are currently selected in the GUI.
	`uniq:REF:/melody/>>`	Picks the labels starting with `/melody/` that target linear visco-elastic modules (`REF`), and also ensures unicity of targeted modules (only one label is kept for each module).
D	`@:/part25/>>`	Picks the permanent labels of the modules having a label starting with `/part25/`
	`#:/gesture1/*`	# orders the picked labels according to the indexes of the terminal contextual names, instead of using lexicographic ordering.

Table 1. Label picker expressions. Single label (A); regular expressions (B); substitution (C) and filtering (D) operators.

We define internal nodes of the label tree as name containers, or *containers*. A container contains contextual names - and can if needed contain multiple contextual names targeting the *same* module. So as modules, containers may be targeted by numerous labels if needed. Containers enable managing (selecting, handling, parametrizing…) all the modules it contains in one shot or, more formally, all the modules that have at least one label starting with one of the container's labels.

Finally, contextual names are indexed within containers. The user labels tree is hence a *rooted and ordered tree*.

4.3 Label Picker Expressions

Label picker expressions (LPE) enable "picking" (or "selecting") multiple labels, and consequently the modules they target, based on their syntactic proximity. Solving a LPE consists in retrieving the list of labels it picks. Most the Tcl commands that form PNSL_Front accept LPEs as arguments. This allows dealing with multiple labels (or multiple modules) in one shot.

Table 1 provides a few LPE and their effects in picking labels, to give a taste of the LPE system. In this table, Part A presents basic examples. Part B illustrates LPE's regular expression syntax. Part C exemplifies "filtering operators", that restrict the picked label. Part D provides examples of "substitution operators", that replace the picked labels with another label for the same module or container, and "ordering operators", that enable managing the order of the picked labels.

4.4 Comparison of the labeling system

The labeling system enables referencing both individual modules and overlapping sets of modules within complex networks. The user can declare as many labels for a module or a container as there are contexts, or editing tasks, in which the module or the container is to be involved. Symmetrically, a module may be named in as many containers as needed. With the labeling system, the modules, and the network, are left unstructured: only the *modules' labels* are hierarchically organized. As for it, the label picker expression mechanism enables picking many labels with a dedicated syntax.

The labeling system can be compared to UNIX/Linux-style file systems: files and their unique inodes, directory hierarchy, and hard links enabling multiple paths for a given inode. Symmetrically, the label picker expression mechanism shares similitude with the regular expressions featured in UNIX shells – these were indeed a source of inspiration in our design. However, hard links in file systems are not very common at the end-user level, whereas, in PNSL, a user usually rapidly defines multiple labels for each module. Other differences are the indexing of names in containers, the handling of integers in labels and in label picker expressions, and the prefixed operators featured in LPEs. Noticeably, the goals pursued with label picker expressions, could also be compared with the principles of Semantic File Systems.

The system shares also similitude with the address patterns featured by the Open Sound Control OSC protocol. A first difference is that OSC does not distinguish *methods* from their unique name in the OSC's address space. Also, the number of entities both systems were designed to deal with differs: usually up to some hundreds in OSC, to hundreds of thousands, possibly millions, in PNSL. Besides the necessity of optimization, this addresses the processing of numbers inside the labels' contextual names.

The XPath query language for XML documents, and also possibly jQuery in the JS/XML world, are other technologies of interest in our context: XPath enables selecting nodes in a hierarchical XML document, where an LPE select labels in the label tree. A difference lies in the fact that XPath manages the object of interest itself (the XML document), whereas LPEs manage labels as means to target modules in the designed network. Also, XPath (and also JQuery) provide more selection possibilities. In particular, XPath selection patterns may not only act over the XML tree, but also on any attribute of any XML node. Conversely, in LPEs, the prefixed operators are so far limited to the type of the targeted modules. This indeed leaves room for possible future improvements, for example toward filtering labels based on modules' parameters or connections.

5. PNSL'S FRONT END

PNSL's front end allows a user to start with PNSL without the need of programming skill. As an example, a simple model could be designed in PNSL without the need of Tcl built in features. Tcl is however available whenever the user needs to go beyond PNSL's basics to start a deeper programming of his modeling process.

5.1 Tcl packages

PNSL_Front splits into 14 packages. Each package groups the commands necessary to handle a phase of the modeling process, or other services: create/delete modules; handle the topology of the masses-interactions network at hand; manage labels and the labels' tree; pick labels by using LPEs; manage physical parameters; handle initial state of the MAT modules; setup audio outputs or the visual mapping of the model during simulation; etc. Thanks to the 14 packages, a host application can restrict which features the user can access in a given interpreter. So doing, the host application may respect and guide the various phases of the user's modeling process. For example, at a given time, an interpreter may handle only the topology of the network, only physical parameters, only initial state, etc.

5.2 Common choices for PNSL_Front's commands

Implementing a domain-specific language on top of Tcl usually consists in extending Tcl with new Tcl commands. PNSL is implemented through 67 such commands. They are usually split in pairs of *getters* to access properties of the model at hand, and *setters* to modify them. Most PNSL commands share a comparable syntax, and most accept one or several LPE (hence: "any number of modules") as arguments. When appropriate, to optimize: commands acting on modules rather than on the label tree, directly compute the LPEs into a list of modules, instead of retrieving full labels; and commands may silently add filtering operators adapted to its semantic.

5.3 Example

To provide a taste of the language, Figure 1 presents a simple PNSL script, translated to English in this paper from the French-based PNSL script. This exemplary script builds and parameterizes a simple GENESIS "linear string", with a few inhomogeneous parameters, very basically excited with an initial velocity.

In this example, line 1 controls the number of masses for the string. Line 2 creates a container in which to build the labels for the new string's modules. Note that the `+` will be substituted by the first integer that has never been used so far after the chosen radix `/string`. Hence, the created container could be automatically named `/string23` if, prior to execution, the label `/string22` had already been defined at some point in the model.

Lines 3 to 6 create the needed modules: masses (`MAS_U`), visco-elastic interactions (chosen linear in this example: `REF_U`), fixed points (`SOL_U`), and audio output (`SOX_U`). The acronyms employed for the module types are those defined by the CORDIS-ANIMA formalism. With the same commands, we decide to declare some initial labels for each created modules. The needed name containers are silently created by the system. The use of `@` (for "use permanent labels") at the end of some of the declared labels denotes that we here do not want explicit contextual names for the modules in the container. For example, a label for the first created mass could be `/string23/mas/@675`, provided `@675` is its permanent label automatically given by PNSL.

```
1  set nbmas 80
2  set root [declare_container /string+]
3  create_module $nbmas MAS_U named $root/mas/@
4  create_module [expr $nbmas + 1] REF_U $root/ref/@
5  create_module 2 SOL_U $root/fixed/@
6  create_module 2 SOX_U $root/audioout/1..
7  declare_label $root/mat/@ for $root/fixed/#1    \
8                $root/mas/* $root/fixed/#2
9  declare_label $root/outmas1.. for $root/mas/#1 $root/mas/#5
10 declare_label $root/excit_point  $root/mas/#2
11 connect_lia $root/audioout/> to $root/outmas*
12 foreach_integer index in 1..[expr $nbmas +1] do {
13   connect_lia $root/ref/#$index      \
14               $root/mat/#$index $root/mat/#[expr $index + 1]
15 }
16 set posH 0
17 foreach_label label in $root/mat/* do {
18   set_bench_position $label bH $posH bV 0
19   set posH [expr $posH + 0,01]
20 }
21 set_param_phys $root/>> pM 1 pK 0,01 pZ 0,0001
22 set_param_phys $root/mas/#2 pM 2
23 set_param_phys $root/outerref/#1 pK 0,02
24 set_initial_state MAT:$root/>> iX0 0 iV0 0
25 set_initial_state $root/excit_point iV0 1
26 print_label $root/>>
27 foreach_label module in @:uniq:$root/>> do {
28   print module $module has labels [get_labels $module]
29 }
30 select $root/>>
```

Figure 1. A simple PNSL script

Lines 7 to 10 declare other labels for some of the modules, enabling to later refer to these modules specifically. For instance, line 7 creates and fills the name container `/string23/mat/` that gathers all the <MAT> (masses + fixed points) of the string.

Line 11 to 15 set up <LIA> - <MAT> physical connections for all the modules of the string. Connections are established here by using the basic `connect_lia` command, but a straightforward feature provided in the PNSL library of scripts could also build such a simple string topology.

Line 16 to 25 set up the modules' properties: position on the GENESIS GUI's 2D topological bench (see [11]), then physical parameters and initial state. For instance, line 21 affects "default parameters" to all the modules throughout the string. Then, lines 22-23 introduce some inhomogeneity, by modifying a few parameters in a few modules. Symmetrically, line 24 puts the model at rest, and line 25 introduces some energy by changing the initial speed of one of the masses.

On lines 26 to 29, we print a few information about the created string and its modules' labels. On line 27, the ESL `@:uniq:$root/>>` selects the permanent labels of all the created modules (`@` selector), and ensures each of them appears only once (`uniq` selector). Then, the loop prints all the labels of each module.

Finally, line 30 ensures selection of the created modules in the modeler's GUI once the script terminates.

Noticeably, in the above script, keywords such as `for`, `to`, `in` are optional. Also, it would have been possible to omit certain aspects (*e.g.*: omit physical parameters set-up), and let the user design them later in the GUI once the script has been executed. Finally, note that the PNSL library of scripts provides routines that generate full-featured strings, membranes, plates, etc. – which could be employed in real-case situations to generate such a simple model, instead of this exemplary script.

6. IMPLEMENTATION

PNSL is accessible to the user as a stand-alone scripting language, which uses CORDIS-ANIMA simulation engine to simulate the designed models. Besides, thanks to the use of the Front End Pattern, the language is now also incorporated into the interactive modeling software frameworks. This includes the GENESIS environment for sound design and musical composition [11] and the MIMESIS environment for movement and animated image synthesis [10]. In these environments, both the GUI and PNSL now fully rely on PN_Core for their core data structures and features. These creative tools are disseminated through 3 mechanisms: through research and creative cooperative projects, commercially, and through pedagogy/teaching/workshops.

The language itself, PN_Front, is first used internally, as a file format, to save and share models in a human-readable way, and also to manage the undo/redo mechanism in the GUIs.

Second, on the user side, PN_Core features are available both in the language and in the graphical interface. Three embedding mechanisms are available to the user in the GUIs: (1) PNSL is accessible within a command-line interpreter, to fire any PNSL command on demand; (2) It is offered in a script editor, to program, then execute, any modifications of the designed model; (3) It can be used to program new user-defined modeling actions, hence to extend the GUI with new commands specialized for the user's needs. Hence, whenever adequate, the user can employ the language to implement any modeling tasks, as a *vis a vis* of the graphical direct-manipulation and WIMP features of the GUI - see Figure 2. The network of modules (including its topology, its parameter and its initial states, etc.), and all the labels, are transparently accessible and can be edited through any of the two interaction mechanisms, language and GUI. Hence, these environments now fall into the scope of multimodal interfaces (as defined in Human-Computer Interaction, when a given interface gathers multiple "representational" or "interaction" modalities), by enabling cooperation of the graphics-based and the textual-based human-computer interaction modalities within Physics Networks modeling.

7. CONCLUSIONS

This article introduced the Physics Network Scripting Language, PNSL, a domain specific language that extends today's modeling possibilities by allowing a user to design by programming any physics-based model within the physics-network, or masses-interactions, paradigm.

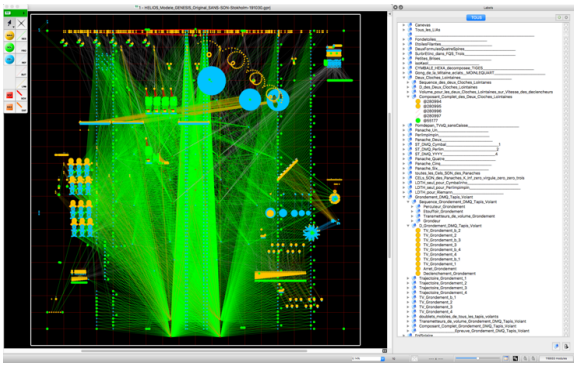

Figure 2: a representation of the musical piece *Helios* by Claude Cadoz, 2015, in the current version of GENESIS equipped with PNSL. *Helios'* model is made of 116 933 elementary modules. Its label tree is rather complex, with close to 1000 overlapping name containers, and a tree-height of 6. On the right corner, the widget displays and enables managing the label tree from the GUI front end. Note that, as explained above, modules are sometimes unnamed in name containers (e.g. `@67`), and sometimes named with an explicit user-defined name.

At a time when, in the field of physics-based simulation, often only preconceived models or large predesigned building blocks are available to artists, PNSL aims at empowering the user (*e.g.* an artist, a musical composer, an animation designer, a researcher…) with the modeling of his/her own physics-based models. PNSL builds on the CORDIS-ANIMA formalization of Physics Networks. Thus, the language may apply to various modeling process rooted on masses interconnected by physical interaction - including mass-spring meshes, and others. PNSL supports modeling no matter the network's topology (regular and irregular), its complexity (from a few to hundreds of thousand modules), its parameters, the dimensionalities of its modules, and the application domain pursued, including Computer Music, Computer Graphics and Virtual Reality, Physics teaching, research, etc.

Since the first release of PNSL, the usage of PNSL developed into various directions.

First, a number of existing models have been re-written using PNSL: large unstructured models (e.g. of non-linear gongs, smoke models…), finely structured models (puppets, vehicles…), models of many sound structures, musical instruments models and past music pieces.

Second, users of GENESIS and MIMESIS have started experiencing with PNSL, for artistic works falling either into the field of musical creation and visual creation, and for research purposes. The feedbacks we obtained show that users that are neither scientists nor programmers are able to start learning and using PNSL, and in particular are able to apprehend the labeling system; and conversely that PNSL enables more complex scripting of modeling tasks. For example, users scripted the generation of large regular model's parts with specifically crafted topology, the computing of the physical parameters of a model to match various desired effects ("inverse problem"), the

management of the spatial organization of a model, the selection of modules in the GUI according to searched properties, etc. Users also reported that the language enabled opening new branches in their creative effort. This concerns for example the design of large networks' parts with specific topology and their articulation with other parts in the model, thin managing of physical parameters' profiles by mathematical laws, and programming of systematic modeling/simulation experimental loops - all of these being difficult to implement with a direct manipulation GUI.

Hence, we hope PNSL is a contribution to the technology serving the trend that envisages physics-based modeling, and not only physics-based simulation, as a core mean to extend creativity with the computer.

Acknowledgments

This work has been supported by the French Ministry of Culture, and the collaborative project DYNAMe funded by the French Agence Nationale de la Recherche. We thank the PNSL beta testers, especially Fabien Henry, Ali Allaoui, Clément Werner, Jérôme Villeneuve and Francisco Huguet; Peter Torvik for the help on the English terminology for PNSL features; and the Tcl community for its assistance.

8. REFERENCES

[1] S. Bilbao: "A family of conservative numerical schemes for the dynamical von Karman plate equations". Numerical Methods for Partial Differential Equations, 24(1):193-216, 2008.

[2] V. Chatziioannou, V., & M. Van Walstijn: "An Energy Conserving Finite Difference Scheme for Simulation of Collisions". Proceeding of the SMC Sound and Music Computing Conference, 2013, Stockholm, Sweden.

[3] C. Cadoz, "The Physical Model as Metaphor for Musical Creation. pico..TERA, a Piece Entirely Generated by a Physical Model". Proceedings of the International Computer Music Conference (ICMC'02), Sweden, 2002.

[4] J. Kojs, S. Serafin, Stefania and C. Chafe: "Cyberinstruments via Physical Modeling Synthesis: Compositional Applications". Leonardo Music Journal. 17. 61-66. 10.1162/lmj. 2007.17.61.

[5] C. Cadoz, A. Luciani, J-L. Florens, "Responsive Input Devices and Sound Synthesis by Simulation of Instrumental Mechanisms: The Cordis System". Computer Music Journal, 8, N°3, pp. 60-73. M.I.T. Press, Cambridge Mass. 1984.

[6] M. Karjalainen: "BlockCompiler: Efficient Simulation of Acoustic and Audio Systems". 114th AES Convention. 2003 March 22-24. Amsterdam. The Netherland.

[7] A. Luciani, "Art kinesthésique et art plastique : contribution à l'émergence d'un art visuel dynamique". Créativité Instrumentale et Créativité Ambiante. Enactive Systems Books Editor. ISBN 978-2-9530856-1-7, 2012, pp.17-63.

[8] N. Castagné, C. Cadoz, "10 criteria for evaluating physical modelling schemes for music creation". Proceedings of the 6th Conference on Digital Audio Effects (DAFX-03), London, UK, Sept 8-11, 2003.

[9] O. Tache, C. Cadoz, "Organizing mass-interaction physical models: the CORDIS-ANIMA instrumentarium", Proceedings of the International Conference on Computer Music, ICMC'09, Montreal, Canada, pp411-414.

[10] M. Evrard, A. Luciani, N. Castagné, "MIMESIS: Interactive Interface for Mass-Interaction Modeling", in Proceedings of Conference on Computer Animation and Social Agents - CASA 2006, Geneva, N. Magnenat-Thalmann & al. editors, pp177-186.

[11] N. Castagné, C. Cadoz, A. Allaoui, O. Tache, "G3: GENESIS Software Environment Update". In proceedings of the International Computer Music Conference (ICMC'09) – Montreal, Canada, August 2009 – pp407-410 - ISBN 0-9713192-7-8.

[12] E. Berdahl, J-O. Smith, "An introduction to the synth-a-modeler compiler: Modular and open-source sound synthesis using physical models", in proceedings of Linux Audio Conference LAC-12, CCRMA, Stanford, 2012.

[13] D. Howard, S. Rimell, "Cymatic: A Tactile Controlled Physical Modelling Instrument". Proceedings of the 6th Conference on Digital Audio Effects (DAFX-03), London, UK, Sept 8-11, 2003.

[14] C. Henry, 2004. "pmpd : Physical modelling for Pure Data". Proceedings of the International Computer Music Conferences (ICMC'04), November 2004, Miami, florida.

[15] A. Van Deursen, P. Klint, J. Visser, "Domain-specific languages: an annotated bibliography". SIGPLAN Not. 35, 6 (Jun. 2000), 26-36.

[16] D.S. Wile, "Supporting the DSL Spectrum". Journal of Computing and Information Technology - CIT 9, 2001, 4, 263–287.

[17] D. Spinellis, "Notable design patterns for domain-specific languages". Journal of Systems and Software, v.56 n.1, p.91-99, Feb 1, 2001.

[18] M. Mernik, J. Heering, A.M. Sloane, "When and how to develop domain-specific languages". ACM Computing Surveys (CSUR), v.37 n.4, p.316-344, December 2005.

[19] J.K. Ousterhout, "Tcl and the Tk toolkit". Addison-Wesley, Reading Massachusetts, 1994, 460 pages.