Finding the jewels in Scaler - a tentative project

v1 12.05.21

Version history

ref	date	who	comments
1	12/05/2021	GTR	Initial 'straw man' document for comment

0 Management summary

0.1 The purpose of this document is to outline possible approaches to help a user of Scaler 2 to manage the vast number of unique musical patterns that the application can create, and in so doing, maximise the creative benefit they may gain from this wonderful product.

0.2 Two suggestions are proposed. The first is wholly independent of the product itself, and has the advantage that the Scaler authors need to make no changes to the application for coexistence; the second approach requires the Scaler authors involvement to make a minor - but wholly transparent - change, not to the application per se, but to the snapshot dump of the system it can produce.

0.3 This parallel proposed application would be able to record details of auditioned musical patterns for query and display, and to be able restore the system states at the point at which user saved them. It is envisaged that the saved patterns could have a user rank applied, and also to have notes and comments attached.

0.4 An outline of these functions is given in paragraph 3.1.

1 Background to project

1.1 Scaler's starting point for developing musical pieces involves 'songs' 'artists' and 'scales', which load what might be described as 'progression sets'. Each of these progression sets can then be overlaid with 'performances' to embellish them, comprising note sequences or rhythmic changes, to create a musical structure of up to 8 bars long.

Multiple of these structures can then be chained together, and a whole range of variation in terms of pitch, voicing etc applied (for the most part) at a bar level.

1.2 Just the simple combination of some given progression set and some given performance results in many *hundreds of thousands* of potential musical fragments, even before they are chained together to form longer sequences, or 'per chord' variations are applied.

1.3 *For someone wishing to create a composition, in the main, there is no obvious a priori mechanism for determining if one of these musical fragments will be suitable for some target piece the composer is seeking to create.* One approach to the challenge of finding candidates is to define a subset of artists, songs and performances which are favoured by the creator's chosen genre(s) (e.g. World, Trance etc) and hence *may* generate something of interest, and audition just those, in combination with some equally subjective subset of performances.

1.4 The approach in 1.3 above - restricting progressions and performances to an initial 'theme based' preference - however ignores the reality that the combination of a rejected progression and a rejected performance might be a jewel awaiting discovery. The assumption that *subjectively* unsuitable constituents implies that the combination thereof would also not create anything worthwhile is to fall into the logical trap that the *"absence of evidence is evidence of absence".*

This note explores alternatives for being able to explore efficiently a wider range of such combinations in order to maximise the creative potential within Scaler.

2 The auditioning challenge - getting the best out of Scaler.

2.1 In the absence of any sound mechanism for reliably a priori determining which combinations from a very large number might form the basis of the composer's target piece, efficient ways of auditioning combinations are required. Such auditioning will probably not be random, but on the basis of prioritising independently progression sets and performances and working through them methodically. Even so, there are likely to be a very significant number combinations to audition.

2.2 By default, the composer might take a list of progression sets and performances (not currently available in hard copy ?) and working through the priority list, mark up (by hand) a short list of candidates to focus on. This can still very time-consuming, and potentially provides only a subset of information the composer might wish. (For example the question "Is there anything else in E Lydian?" etc. would not be easily answered.)

2.3 However, there is already a means within Scaler to provide the framework for a solution to this efficient auditing problem - the 'export state' function.

3 Framework for an auditioning / reporting solution with the 'export state' file as a basis

3.1 What would simplify and enhance this tedious process is

(a) a rapid and effective mechanism to save the identity and details of some candidate 'composition base' i.e. the user's starting point,

(b) a means to add that choice to a persistent collection of same (which for convenience will henceforth be referred to as a 'database'),

(c) ideally, a means to append to the data in the database the user's rating of the combination and potentially, 'notes' on the choices,

(d) a means to report certain property values from that database, and,

(e) a means to query the database.

3.2 By definition (since Scaler can restore the application to the point in time the state export was performed) all the information available at the time of the save resides in the 'export state' file. However, although this is ostensibly cognisable to the human eye (being in XML), it is not in the most part in a form relatable by the user to the music auditioned; the names of songs or artists, for example, are encoded and not recognisable.

3.3 It follows therefore that either

(a) before such encoded data are added to the persistent store referred to in 3.1 it has to be mapped to a form recognisable to the user, or,

(b) the encoded data are persisted in the form held in the 'export state' file and then transformed during operations as at 3.1(d) and 3.1(e) into a user cognisable form.

For reasons explained in the technical notes herein, the latter would seem preferable from a design perspective.

3.4 This system could be implemented without any change to Scaler or using valuable resources of the Scaler team. However, for reasons already communicated to the Scaler team, it would be made rather easier by two simple additions to the state file, being the timestamp forming part of the export state filename and the XSD schema reference.

3.5 Hence, the user application would list or display summary information from selected auditioned Scaler combinations, showing such things as song, artist, key and other values, together with a possible user assigned ranking score and some text notes. This application could either (a) seek to locate the source XML export state file in some defined location by means of the timestamp or (b) have the export state data stored in the database as a BLOB. This latter approach opens the scope to all manner of future developments, since the full data of the save image would be available for further analysis.

3.6 Simple searches could respond to use queries such as "what did I like in E Lydian" or "what performances did I choose with song Trance 1" (or of course 'trance *').

3.7 An initial schema for the database will be issued shortly.

4 What this project is not

4.1 Just as important as setting out the goals for what a project is intended to deliver, common sense (coupled with bitter experience) dictates that it is sound to say what are the goals are not.

4.2 Assume someone is working on a Scaler project and they make changes, although they could move everything to a DAW and then assume that to be the master working area, a more practical approach would be to export the state of a project, and then restore that last export state file when the next phase of development of the piece. How to identity where the project was saved amongst a number of work in progress files is thus a key issue. This could either be saving this with a recognisable name, or using the default name and saving to a project specific folder. In this latter scenario, the user would simply restore the latest time stamped state file on returning to the project.

4.3 This highlights the apparent Markovian nature (no prior memory other than current state) of the export state file. It has nothing eye cognisable in it to unambiguously define its origins or its relationship to any other state export. That can only be done indirectly by the alternatives in 4.2, namely textually from the file name or physical segregation from other saves by a folder.

4.4 So this project is not intended to (nor can it as currently envisaged in this first instance; however, see paragraph 7) provide any information about the temporal history of the musical piece being

restored. ***Its sole purpose is to provide a set of potential candidates of sequence / progression combinations for future working on from an otherwise unmanageable number of progression / performance variants.*** Once one of these candidates has been chosen on as the basis for a piece, other methods are needed to manage the evolution over time of that piece, in the normal manner of working on such projects.

4.5 Paragraph 7 discusses potential feature changes which might facilitate the management of a piece over time, but it is assumed that the Scaler developers will maybe have that on their to-do list anyway.

5 Technical observations and assumptions

5.1 The author has no knowledge of the inner workings of Scaler, and hence all the following statements and assertions are on the basis of reasonable (?) inference.

5.2 A given export state file is unique to a local environment in 'identity' (as in that which makes the file non-fungible, in this case given by the time stamp in the file name) from a practical rather than theoretical perspective. [It is potentially not unique when the universe of Scaler instances are considered.] However, the content is not unique; as a simple example two state saves could be performed without making any changes to the target piece between them and the resulting file content (since the timestamp is not contained within it) would be bitwise identical.

5.3 Nevertheless, it would still be potentially valid to make the timestamp the primary key in the envisioned database. This would essentially make it 'write once'; data records would always be appended. A suggestion which has been made to the Scaler authors to add the time stamp to the exported state, to avoid having to pick this from the file name during the processing,

5.4 A number of essential (for reporting with human cognition) Scaler property values are encoded as UUID's in the export state file, which appear to be version 4 UUIDs, since there would be no obvious rationale for making them version 1, 2 or 3. Following the approach of 3.3(b) these would be written to the database, and then transformed later, either during initial processing of writing the database, or subsequently triggered by some external request , or lazily, on demand on the fly.

5.5 Only a small number of UUID values need to be converted to the associated property values to fulfil the goals of the project. This is because these would be the properties for the user to identify and assess the usefulness of the persisted states. The actual UUID values can in the main be determined by changing some property value in Scaler and exporting the state before and after the change and hence building up a list of correspondences of UUID and property name, and adding them to a key (UUID) / value (property name) pair list. Winmerge or some other similar utility can then highlight differences in the XML in the state file, thus revealing the values.

{It may be that the Scaler authors would publish these in which case this tedious task would not be needed.}

5.6 Since UUIDs are application unique, (in the sense that there would be no UUID synonyms in a export state file for all of the UUIDs contained therein, regardless of the Scaler property to which they were assigned) so the system would not have to differentiate between the property types represented by the UUIDs and hence only a single key / value pair list of UUIDs linking to any set of properties is required to generate the UUID to property name mapping for user visibility. In other words, there does not need to be a separate key/value pair list for different properties such as scale key, or expression set. This simplifies the design somewhat.

5.7 A likely tool to effect the end-user utility would be Talend Studio, which has an open source version. This would map the state file to a normalised (or more accurately, flattened) store of relevant properties, (together with any data such as a user rating or notes) probably in SQLite.

6 Alternate approach

6.1 The approach to creating a database in prior paragraphs is predicated on the assumption that no changes would need to be made to the export state file (but refer paragraph 3.4) and the solution proposed would have no dependence on Scaler staff. However, certain changes which could be made by the Scaler authors would not only simplify the database logic, but also enhance he functionality the database could provide.

6.2 If the Scaler authors could modify the output file so, as well as the UUIDs, the 'native' form of the data was written to the export state file (i.e. the property value associated with the UUID value), this would remove the complexities of mapping UUIDs for the suggested user application, and it would become relatively trivial to effect the functionality needed. It is assumed that the 'native' values exist in Scaler at the time of the export state action is invoked, since the UUID's have had to have already been decoded to display the human cognisable equivalents on the application interface.

6.3 Ideally, this modification needs to be transparent to the Scaler code parsing the file on re-reading this (i.e. the' import state' function) , to minimise any additional maintenance effort for the Scaler coders, and also to avoid having to add additional XML elements or attributes if further UUID expansions were requested. A rather hack (but effective) solution to this would be by using annotations. These can then be picked up by regex in the database update / report / display user function.

An example might be as follows, using Scaler's camel case element or attribute names :

```
xs:element name="ScalerState">
- <xs:annotation>
<xs:documentation>set\selectedBrowserTab="aaa"\noteFilter="aaa"\selectedScale="aaa"\</xs:doc
umentation>
</xs:annotation>
```

6.4 Changes or additions to these data are thus completely independent of any Scaler logic, avoiding imposing any material coding / testing - the additional data are wholly transparent to the Scaler application code at all times.

6.5 The database application would pick up and persist the values using regex expressions.

7 Chain gang - possibility of future capability.

7.1 It may well be that the following possible functions are already I the plans, knowing how forward thinking the Scaler team are. However, the fact that the user database envisioned in these notes is potentially user extensible, may mean that it might be possible to offer some functions which would be not practical / appropriate for a package product.

7.2 Exported state files have no memory, in that there is no obvious way to determine if two files were connected, in that one might have been the evolution of another. So it is not possible to distinguish between 3 export state files which had been produced independently of one another, or if the 3 files resulted from 3 serial exports during one session or series of sessions - i.e. temporal snapshots.

7.3 As a result, since all three records would have been persisted in the user database, and a query would show all three, when in fact the user was only interested in the last one if they originated from serial updates. Of course, the user could delete prior saved files when performing such serial changes, but there might have been a material time gap between two saves, making this task much more reliant on human identification to chose latest state for common snapshots.

7.3 This is the common 'anonymity' issue of distinguishing between 'identity' and 'identifying'. The 3 files all have *unique identities*, but it is not clear if they share a common heritage, and have no identifying anchor for that heritage.

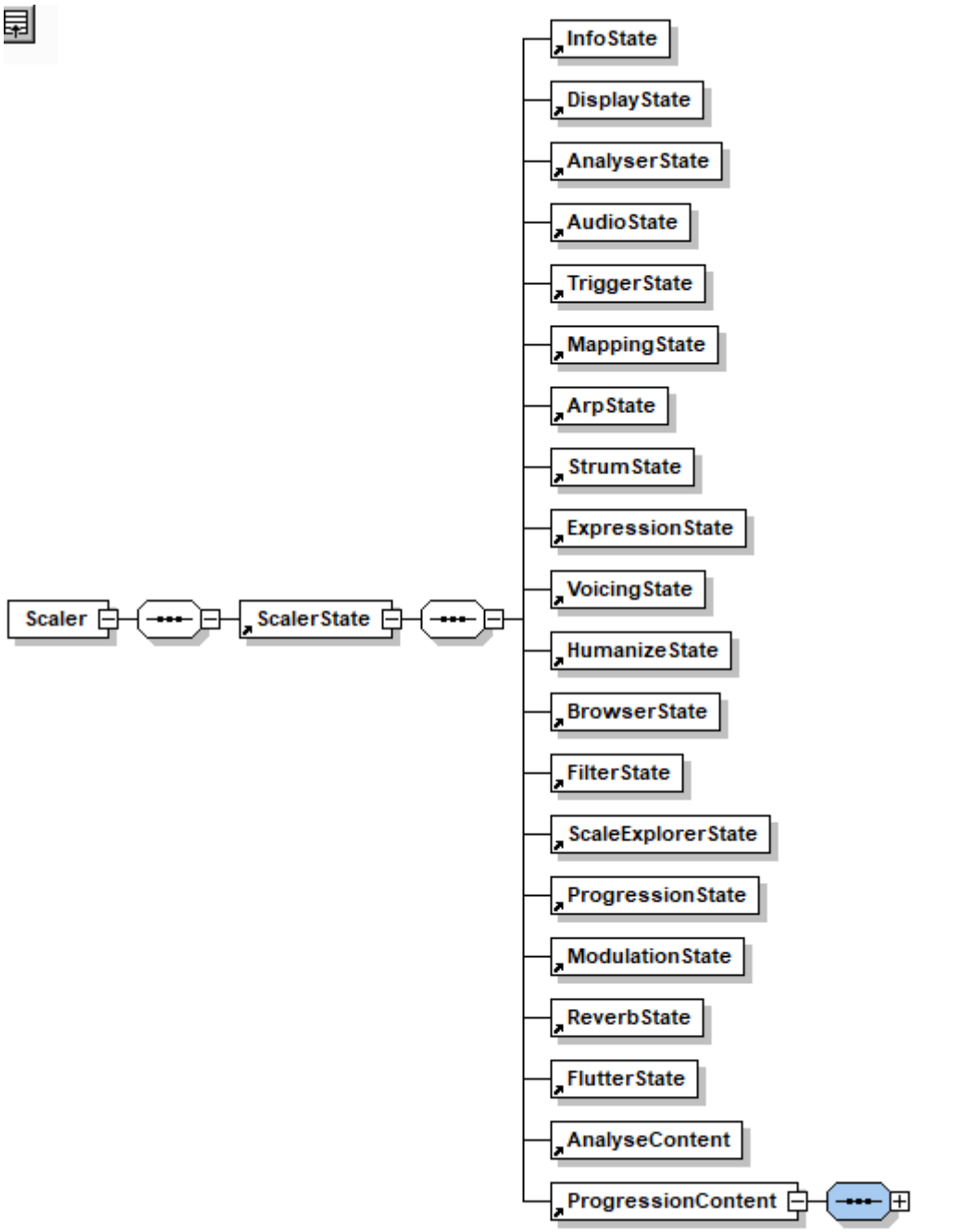
7.4 One way to deal with this is to add a property "parentID", to convey whether a saved session was a 'cold start' or was part of a chain, indicating derivation. By adding this element, and assuming the timeStamp had been added per the original suggestion, the global uniqueness of the time stamp can be exploited.

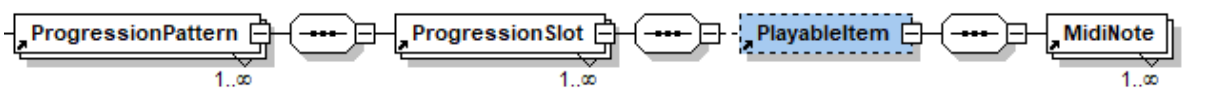
7.5 Clearly the Scaler application can differentiate between a cold start and session information obtained from restoring a prior state. In a cold start situation if an export state is invoked the application the parentID can be set to some null values (spaces or NUL) and the time stamp set as it would be applied to the file name. If an import state operation is invoked, the parentID can be assigned the time stamp so assigned, and if a further export state is preformed, the timestamp set to the new value. This means that serial snapshots from a piece being worked on are linked on a temporal basis, providing on demand PIT ('point in time') recovery.

Crucially, this would allow a referential self-join to follow the evolution of state saves in the database, so the user could see the saved evolution and back track if required. By way of reminder of earlier comments, it would be intended to store each export state file as a BLOB in the database, so there would be no need to have saved the state files elsewhere; they would just be unpacked and then restored as normal into Scaler.

APENDIX I - Derived XSD schema of the export state file

A1.1 The following is an outline derived schema from the export state file. It will (clearly) not contain the full metadata in the actual schema for the product (e.g. enumerations, derived types, restrictions etc.) but it does allow a mapping and transformation application to parse the tree and generate the necessary Java for the database update process.
The text boxes following the diagrams show those elements with more than one attribute.





element **VoicingState**

diagram	**VoicingState**
used by	element ScalerState

attributes

Name	Type	Use	Default	Fixed	Annotation
isVoicingPreviewOn	xs:boolean	required			
lowestNote	xs:byte	required			
highestNote	xs:byte	required			
voicingProfileId	xs:byte	required			

element **BrowserState**

diagram	**BrowserState**
used by	element ScalerState

attributes

Name	Type	Use	Default	Fixed	Annotation
selectedBrowserTab	xs:string	required			
chordSetUUID	xs:hexBinary	required			

element **MidiNote**

diagram	**MidiNote**
used by	element PlayableItem

attributes

Name	Type	Use	Default	Fixed	Annotation
number	xs:NMTOKEN	required			
velocity	xs:byte	required			

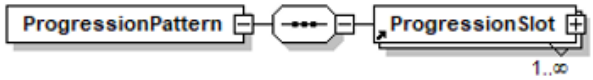
element **ProgressionContent**

diagram	**ProgressionContent** — **ProgressionPattern** (1..∞)
children	ProgressionPattern
used by	element ScalerState

attributes

Name	Type	Use	Default	Fixed	Annotation
selectedTabNum	xs:byte	required			
isEditingIndex	xs:byte	required			
isEditingTabNum	xs:byte	required			
isLooping	xs:boolean	required			

element **ProgressionPattern**

diagram	
children	ProgressionSlot
used by	element ProgressionContent

attributes

Name	Type	Use	Default	Fixed	Annotation
tabIdx	xs:NMTOKEN	required			
tabName	xs:string	required			

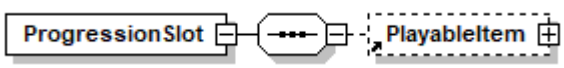
element **ScaleExplorerState**

diagram	
used by	element ScalerState

attributes

Name	Type	Use	Default	Fixed	Annotation
diatonicShapes	xs:string	required			
isOpen	xs:boolean	required			
selectedVoicingUuid	xs:hexBinary	required			
selectedDisplayMode	xs:string	required			
selectedNotePitch	xs:byte	required			

source <xs:element name="ScaleExplorerState">

diagram 

children **PlayableItem**

used by element **ProgressionPattern**

attributes

Name	Type	Use	Default	Fixed	Annotation
octave	xs:byte	required			
inversion	xs:boolean	required			
semitoneDelta	xs:boolean	required			
durationCoefficient	xs:decimal	required			
repeat	xs:boolean	required			
expressionSetUuid	xs:hexBinary	required			
expressionResolutionId	xs:byte	required			
expressionPhrasePlayStyle	xs:byte	required			
expressionPhrasePlayMode	xs:boolean	required			
expressionScaleId	xs:string	required			
arpTiming	xs:byte	required			
arpPatternId	xs:boolean	required			
arpNoteLength	xs:byte	required			
arpOctaveRange	xs:boolean	required			
strummingProfileId	xs:boolean	required			
strummingDirectionId	xs:boolean	required			
playbackPerformanceMode	xs:byte	required			
groupId	xs:boolean	required			
groupPlaybackBehaviour	xs:byte	required			
idx	xs:NMTOKEN	required			
tabIdx	xs:NMTOKEN	required			

APPENDIX II - Accumulated state file enquiry and reporting