Suggestions for the Integra Environment - Update.

Jamie Bulloock, Henrik Frisk

April 18, 2006

Summary of IRC meeting April 7, 2006

participants: Jamie Bullock and Henrik Frisk

The main thread in the meeting was what is referred to in the proposal by Malmö Academy (`http://www.integralive.org/Members/henrikf/suggestions-for-the-integra-\environment/folder_contents`) as the 'score' file. In other words the structure of the data that is passed to modules by value. It seems as if there has been no objections against using an XML file format for this purpose. The discussion was only concerned with terminology and structure and not at all about implementation - how events/sequences/structures are to be read/written/scheduled/dispatched.

The idea of splitting this data over different files, perhaps even one per module, was discussed (this is the approach used by Jamoma). This would mean that each module had its own list of possible 'states' and 'behaviours' associated with it. These could be static (as in a preset) or they could be gestures or patterns that change over time. They could also affect or alter the behaviour of other modules. These XML files act as 'containers' of information about module behaviours. Metaphorically speaking these are the 'players' of the system. These 'players' can be nested into 'groups' in which they interact with each other following a 'conductor' which is a container of meta information about the music (basically the score).

The 'player', the 'group' and the 'conductor' all have the same function, but at different layers of the system. It is anticipated that they would all support the same basic data types. Examples might include, scalars, lists, arrays and hashes. However, a container should only use types supported by the module with which it is associated. Players, groups and conductors could be thought of as instances of the container class. See fig. 1 for an approximation of how the concept might work.

The advantage of using an approach in which this data is split up in many files is that a set of 'behaviours' can be stored and updated along with a module; 'the player' (the XML container) has an instrument (the synthesis module) and knowledge about how to play it (the data in the container). Technically, parsing the data will also be easier and faster compared to a solution in which all of this data is stored in one big file.

The possibility of integrating SMIL was also discussed. Though there are interesting possibilities with this approach, it was decided that the main priority now is to consolidate a basic XML specification for the containers so we can begin development. SMIL could then be incorporated at a later stage and form a subset to the Integra XML container specification.

Following is an example of a module specific XML file. In the example it is assumed that it's possible to interpolate between all the parameters in a module's preset using the

same mode (linearly, logarithmically...). One issue that has to be further discussed also in relation to the OSC namespace is voice instantiation. Should it be assumed that all synthesis modules is capable of producing multiple voices and in that case, what is the procedure for communicating control data to just one of these voices?

The XML file structure has its head and body tags. In the body tag there are three major sections:

- The `preset` tag.
- The `current` tag.
- The `synchro` tag.

The contents of the `preset` tag communicates data *to* the module, the `current` tag receives information *from* the module about its current state and the `synchro` tag synchronizes the sending of data from the player container to the module. The contents of the `current` tag can also be referenced from the `synchro` tag.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<integra xmlns="http://www.integralive.org/DTD"
         xmlns:score="http://www.integralive.org/DTD-extensions/scoreDTD">
  <head>
    <!-- Documentation and instance name and class path of the
         module.
      -->
    <module class-path="/Integra/GeneralAudioModule/GeneralSynth/SimpleSynth"
            instance="smplsnt"/>
    ...
  </head>

  <body>
    <preset>
      <pattern id="1">
        <!-- Holds all the parameters for one (1) static 'preset' or
             one or several notes. Calling this is the equivalent of
             calling a program on a synthesizer.
          -->
      </pattern>
      <pattern id="2">
        <!-- Holds another preset. -->
      </pattern>
      <pattern id="3">
        <!-- Holds another preset. -->
      </pattern>
      ...
    </preset>
    <current update-interval="1s" N="20">
      <!-- Holds the current and last 'N' states of the module with a
           resolution of 1 second.
        -->
      <pattern id="crnt_1"
               frq="200"
               ... />
      <pattern id="crnt_2"
               frq="230"
```

```
            ... />
    <pattern id="crnt_3"
            frq="290"
            ... />
    ...
  </current>
  <synchro>
    <!-- Calling "seq_1" starts playing back the events contained within
         the seq tags. Calling "par_1" starts playing back the
         events contained in the par tags. Calling "event_n" plays
         that one event.

         In the example below event_1 calls pattern 1 and
         interpolates between this pattern and pattern 2 over the
         course of 4 seconds. event_2 calls pattern 1 and
         recursively re-triggers par_1 11 times. event_3 does
         interpolation between pattern 3 and 1 with an offset of 2
         seconds (The offset should be ignored when the event is
         called outside of a seq or par).

         In the next seq the current states are called which allows
         for a different kind of recursion.
    -->
    <seq id="seq_1" offset="0s">
      <par id="par_1" offset="0" recursion-depth="11">
        <event id="event_1"
               length="4s"
               pattern-start="1"
               pattern-end="2"
               interpolation="lin" />
        <event id="event_2"
               offset="1s"
               trigger="/Integra/GeneralAudioModule/GeneralSynth/SimpleSynth.smplsnt::par_1"
               length="4s"
               pattern-start="1" />
      </par>
      <event id="event_3"
             offset="2s"
             length="4s"
             pattern-start="3"
             pattern-end="1"
             interpolation="log" />
    </seq>
    <seq id="seq_2">
      <event id="seq_2::event_1"
             trigger="crnt_1" />
      <event id="seq_2::event_2"
             trigger="crnt_2" />
      ...
    </seq>
  </synchro>
 </body>
</integra>
```

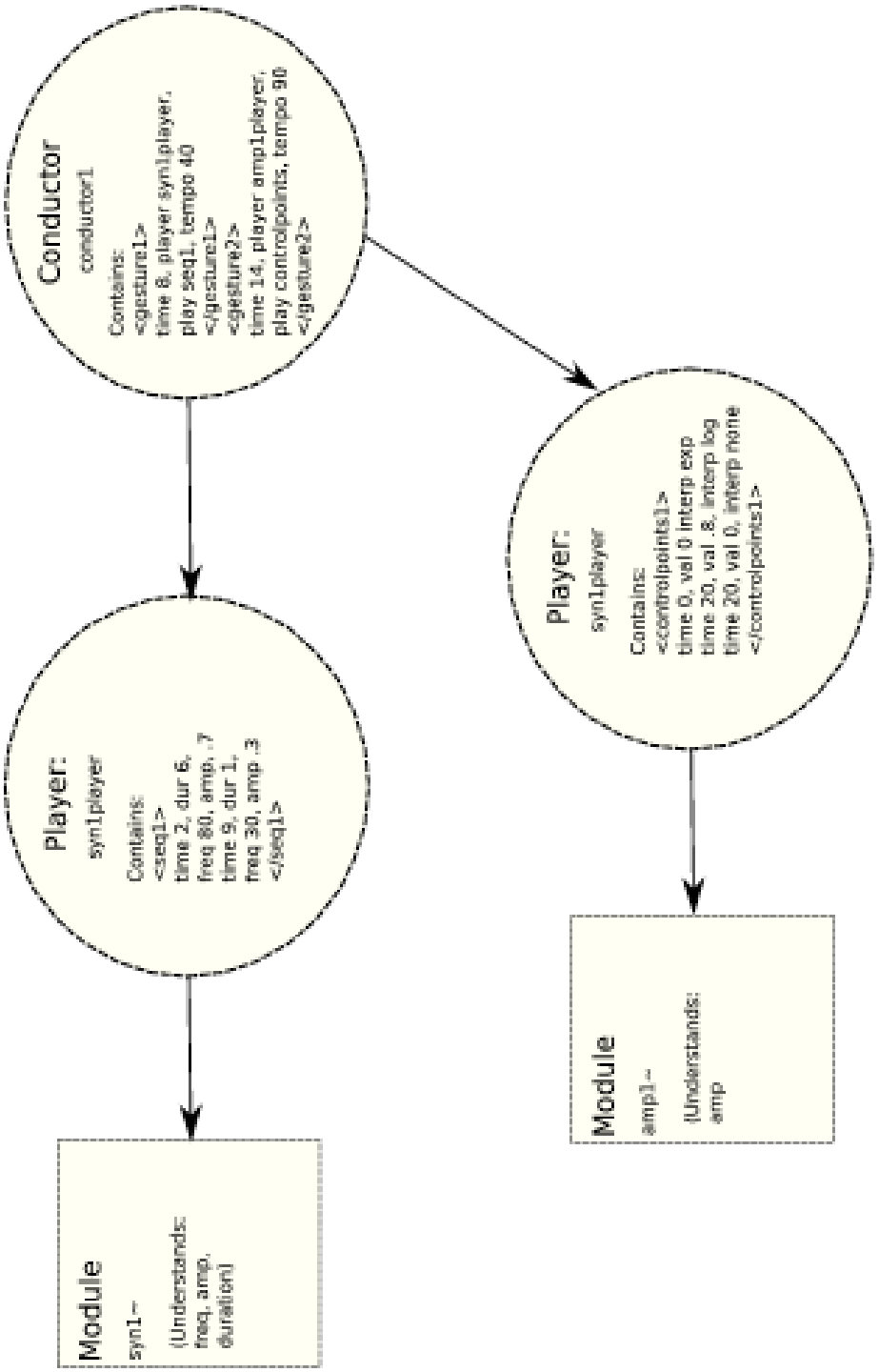

Figure 1: A conceptual overview of the conductor-player-module model.