Graphical Temporal Structured Programming for Interactive Music

ABSTRACT

The development and authoring of interactive music or applications, such as user interfaces for arts & exhibitions has traditionally been done with tools that pertain to two broad metaphors. Cue-based environments work by making groups of parameters and sending them to remote devices, while more interactive applications are generally written in generic art-oriented programming environments, such as Max/MSP, Processing or OpenFrameworks. In this paper, we argue about the specific issues that arise in such environments, and we present the current version of the i-score sequencer. It is an extensive software suite that bridges the gap between time-based, logic-based and flow-based interactive application authoring tools. This is done in a single cohesive graphical user interface, built upon a few simple and novel primitives that give to the composer the expressive power of structured programming, in a time line adapted to the notation of parameter-oriented interactive music.

1. INTRODUCTION

1.1 Temporal multimedia programming

[1][2] - voir mails envoyés à théo, myriam, etc. - langages visuels - références anciens articles i-score - protocoles ? - dire qu'il n'y a pas directement de support des données musicales. - dire que use of interactive triggers a été documenté pour gérer l'interaction, et qu'on s'en sert ici pour faire du flot de contrôle interne à la partition interactive.

2. SYNTAX

The syntax and graphical elements used in i-score as well as the execution semantics are for the most part introduced in [3, 4]. Multiple execution semantics for the same graphical formalism exist. They are based on Petri Nets, Time Automata and Reactive languages.

The novelty lies in the introduction of temporal loops, and of a computation model based on Javascript. These two features allow complete expressive power in the written scores.

2.1 Variables

Variables can be easily implemented by - Variables ? -¿ can be stocked in the tree -¿ no notion of scope - JS can be used for local variable concept

2.2 Temporal structured programming

Structured programming is a programming paradigm which traces back to the 1960's, and was conceived at a time where the use of `GOTO` instructions was prevalent in computing, which often led to code difficult to understand.

The structured programming theorem[5, 6] states that any computable function can be computed without the use of `GOTO` instructions, if instead the following operations are available :

- Sequence (`A` followed by `B`),
- Conditional (`if(Pred) then A else B`),
- Iterative (`while(Pred) do A`).

Additionnaly, the ability to perform computations is required in order to have a meaningful program.

In order to allow for interactive musical scores authoring, we simulate these concepts in the time-line paradigm. A virtual machine ticks a timer and makes the time flow in the score graph. During this time, multiple processes, which include scenarisation, looping, and multimedia processes, are run at each tick.

Processes can be of two kinds : temporal or instantaneous. Temporal processes can be explained as functions of time that the composer wants to run between two points in time. For instance : *do a volume fade-in from t=10s to t=25s*.

Instantaneous processes are functions that will be run at a single point in time. For instance : *send a single random number to an OSC address*.

2.2.1 Scenario

The scenario is an organisation of the elements of the i-score model : time constraint (a span of time which contains temporal processes), time node (which will synchronize the ending of time constraints with the happening of an external event such as a note being played), time event (a condition that will cause its following time constraints to start if it is true), and state (which contains data to send and instantaneous processes). They are shown in fig. 1. The time flows from left to right as in most traditional sequencers, however due to the presence of interactivity, we cannot show all the possibilities of execution of the score, since they would be almost infinite. Hence dashes are shown when the actual execution time is not known beforehand. For instance : *play a D minor chord until a dancer moves on stage*.

In the context of a Scenario, these primitives allows for sequencing elements, conditional branching, and interactive triggering, but are not enough for looping. The GUI of the i-score software allows for all the common and expected operations for elements of a scenario : displacement, scaling, creation, deletion, copy-paste...

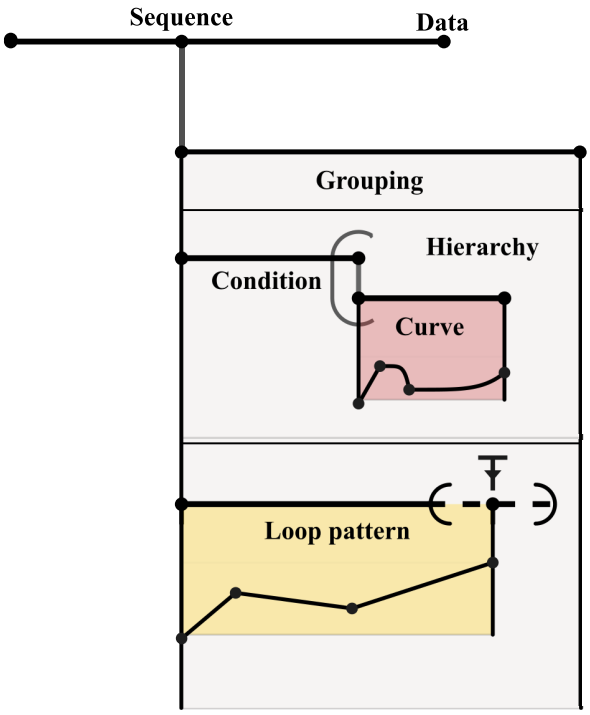

Figure 1. Screenshot of a part of i-score, showing major elements of the formalism. The time constraint is the full or dashed horizontal line, the states are the black dots, the time nodes are the vertical bars, and a time event is shown at the right of the "Condition" text. Interactive triggers are black **T**'s with a downwards arrow. There are five processes (capitalized) : an englobing Scenario which is the hierarchical root of the score, another Scenario, in the box "Hierarchy", an Automation on a remote parameter in the "Curve" box, a Loop in the box containing the loop pattern, and another Automation that will be looped.

2.2.2 Loop

The loop is another organization of the same elements, more restrictive, and with a different control flow : it is composed exclusively of two time nodes, two time events, two states, and a time constraint in between. When the second time node gets executed, the time flow is reverted to before the execution of the first time node. This means that if the composer adds an interactive trigger on any of these time nodes, the Loop will be able to do pattern of different durations at each loop cycle. This is strictly more general than loops in more traditional sequencers such as Avid Pro Tools, Steinber Cubase or Apple Logic, when the loop is generally a strict duplication of the audio or MIDI data.

2.2.3 Communication

The software mainly communicates via the OSC [1] protocol, and maintains a tree of parameters able to mirror the object model of remote software built with Max/MSP, PureData, Unity3D or any OSC-compliant environment. Conditions, interactive triggers, and more generally all processes of i-score have unrestricted access to this tree, and can read its data, perform computations on it, and write it back to the remote software. Music can currently expressed with the help of either a remote software or hardware device that is able to convert OSC messages to either MIDI or audio. One can for instance easily build a PureData patch to convert OSC messages matching a certain pattern to the corresponding MIDI notes.

3. AUTHORING FEATURES

i-score provides multiple authoring features aiming to allow the composer to express himself. Besides, the software is based on a plug-in architecture which allows for easy extension of its capabilities.

- Javascript support : one can use Javascript scripts both as temporal and instantaneous processes. When writing a temporal process, the composer has to provide a function of the following form :

```
function(t) {
    var obj = new Object;
    obj.address = '/an/address';
    obj.value = 42;
 return [ obj ]
}
```

  This function will get called at each tick with the time `t` being a floating point between 0 and infinity, 1 being the default date that the composer has set. The messages returned will be applied to the local state of the tree and sent remotely if corresponding nodes were found. When writing an instantaneous process, the function to provide is similar, but does not take a time argument. Functions writer are encouraged to write stateless functions, since it would allow for optimizations opportunities such as precomputing an array of values.

[1] Open Sound Control

- Automation : the traditional DAW automation, which writes in a parameter over time. Currently provided are 1D (linear, power) and 3D (cubic spline) automations.
- Mapping : similar in appearance to the automation, this process will take an OSC address as input, apply a transfer function, and write the output to another address.
- Recording : one can record either automations which will be able to apply automatic interpolation for numeric parameters, or record any kind of input message to replay it afterwards.
- Execution speed control : the temporal constraints all have a multiplicating coefficient for execution speed.
- Introspectability : i-score exposes the current score in the same tree that the remote devices. Multiple attributes can be queried, and to some extent modified during the execution of the score. For instance, the current position of the play head, the activation status of interactive triggers, and the execution speed of all the temporal constraints are controlable parameters. The elements are organized hierarchically according to their hierarchy in the scenario, and can also be controled from the network. Plug-ins can expose their own attributes in this tree.
- Interactive execution : during the authoring process, it is sometimes necessary to replay just a part of the score. i-score provides this capacity. However, it is necessary for the composer to choose beforehand the truth value of conditions he wants to test. There are two ”debug” execution modes : one that will directly start playing from where the user points the mouse, and another that will try to merge all the previously sent values, to put the external environment in the same state that it would be if the score had been played normally up until this point. This breaks if there are physical processes that are not instantaneous, as would be the case for smoke machines for instance.

4. TEMPORAL DESIGN PATTERNS

In this section, we present various design patterns that can be used when one wants to build an interactive score with i-score. We will showcase event-driven scores, which will have a behaviour similar to a traditional computer program executing instruction after instruction at maximal speed, and an example in simulating the concept of function in a time-driven model.

4.1 Event-driven design

Event-driven, or asynchronous design is a softwar design paradigm that is centered on the notion of asynchronous communication between different parts of the software. This is commonly used when doing networked operations.

Instead of writing : `play A; play B; play C`, in event-driven programming, one would write :

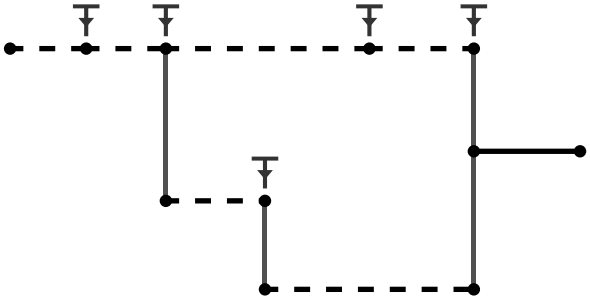

Figure 2. An example of event-driven score

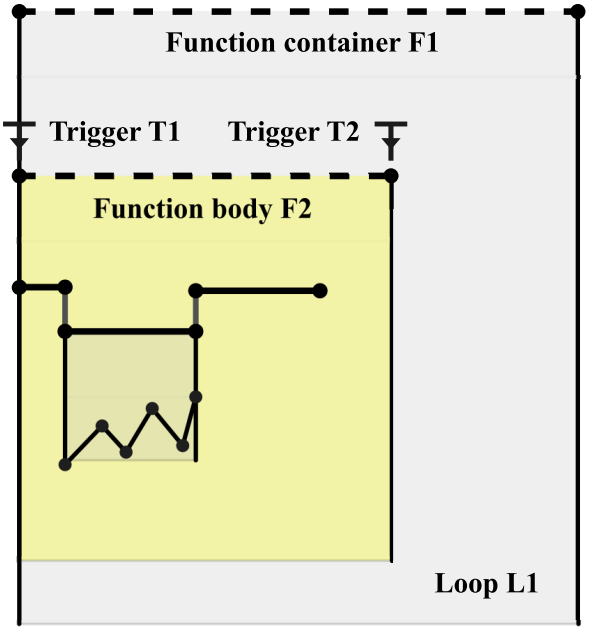

Figure 3. Implementation of a function in i-score

```
when A is triggered
    play B
when B is triggered
    play C
play A
```

Since i-score supports interactive events, one can easily write such event chaining. An example is given in fig. 2. The advantage is that ordered operations can be written easily : there is no possibility for B to happen before A if there is a time constraint between A and B. However, the execution engine will introduce a delay of one tick between each call. The tick frequency can be set to as high as one kilo-hertz. Dependencies can also be graphically explicited : here, the last time constraint will only be executed after all the incoming branches were executed. This allow to write a score that says ”start section B five seconds after musician 1 and 3 have stopped playing”. There is no practical limit to the amount of branches that can be synchronized in this way.

4.2 Simulating functions

The notion of function is extremely common in programming languages. It consists in an abstraction around a behaviour that can be called by name easily. The concept of

function is useful, because it means that if one wants the same behaviour to occur at multiple times during the execution of the program or the score, this behaviour will have to be written only once. However, the visual flow becomes destructured, because the definition of the function is given at a point in the program / score, and its usage is at other points.

We present in fig. 3 how one could create a simple notion of function that will then be able to be recalled at any point in time. However, it suffers from a restriction due to its temporal nature : it will only be able to be called when it is not already currently running. This is due to the single-threaded nature of the execution engine : there is only a single playhead for the entirety of the score, hence it cannot play the same data at two different times.

The function is built as follows :

- A time constraint in the root Scenario will end on an interactive triggering set with infinite duration.
- This time constraint contains a Loop process. The functions is named `f` in the local tree. The interactive triggers T_1, T_2 at the beginning and end of the pattern time constraint are set as follows :
  - T_1 : `/f/call true`.
  - T_2 : `/f/call true`.

  A state triggered by T_1 should set the message: `/f/call false`. This is so that the function does not loop indefinitely and has to be triggered manually again.
- The loop contains the actual "function", that is, the process that the composer wants to be able to call from any point in his score.

- additivity of data

5. MUSICAL EXAMPLE : POLYVALENT STRUCTURE

In this example, showcased in fig. 4, we present a work that is similar in structure to Karlheinz Stockhausen's *Klavierstück XI* (1956), or John Cage's *Two* (1987).

The complete work contains variables in the tree, as explained earlier, and a temporal score.

The tree is defined as follows :

```
/part/next [int = 0]
/part/1/count [int = 0]
/part/2/count [int = 0]
/part/3/count [int = 0]
/exit [bool = false]
```

The score is as follows : there are multiple muscial parts containing recordings of MIDI notes : **Part. 1, 2, 3**. Thes parts are contained in a Scenario, itself contained in a Loop that will run indefinitely. At the end of each part, there is an orange State that will write a message "true" to a variable `/exit`. The time constraint of the loop ends on an orange interactive trigger, T_{loop}. The Loop itself is inside a Time constraint ended by an interactive trigger, T_{end}. Finally, the parts are started by interactive triggers $T_{\{1,2,3\}}$.

The conditions in the triggers are as follows :

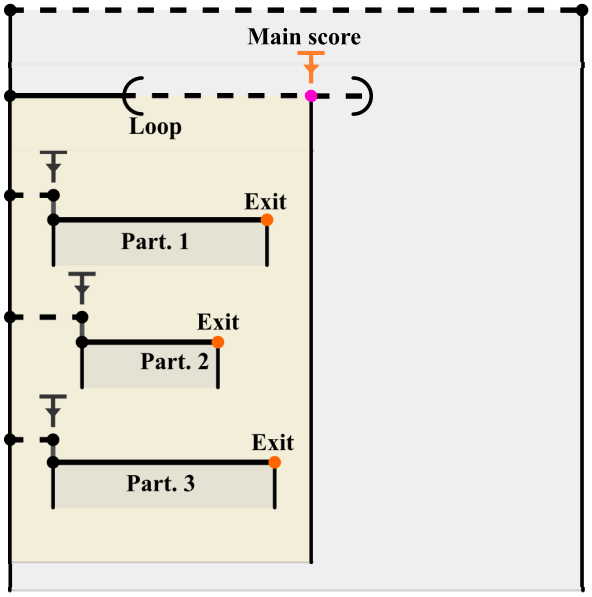

Figure 4. A polyvalent score in i-score

- $T_{\{1,2,3\}}$: `/part/next == {1, 2, 3}`.
- T_{loop} : `/exit == true`.
- T_{end} :
  `/part/1/count > 2 or`
  `/part/2/count > 2 or`
  `/part/3/count > 2`.

Finally, the pink state contains a Javascript function that will draw a random number between 1 and 3, increment the count of the relevant `/part`, and write the drawn part in `/part/next`. If any count becomes greater than two, then T_{end} trigger will stop the execution : the score has ended. Else a new loop iteration is started, and either T_1, T_2 or T_3 will start instantaneously.

Hence we show how a somewhat complex score logic can be implemented with few syntax elements.

Another alternative, instead of putting MIDI data in the score, which makes it entirely automatic, would be to control a screen that displays the part that is to be played. A pianist would then have to interpret the part in real-time which would bring back the human aspect.

6. TOOLING

There are plenty of parts that requires new problems to be solved in i-score, especially in the area of debugging. For instance, debugging is generally used to understand and find out easily why did not a program produce the expected result. In i-score, there is not too many problems due to low-level memory management and there cannot be crashes of the interpreter, minus potential bugs in its implementation. However, as soon as the show's logic becomes complex, it is useful to have tools that give the authors the possibility to pinpoint where the execution of the show does not match the expected behaviour. Since the presented formalism has extensive expressive capabilities, the composer might find himself confronted to the same family of problems that programmers encounter daily. Nowadays,

many programming language offer very robust tooling, as can be seen in the performance tools that are provided in integrated development environments such as Xcode, Visual Studio, and the various Unix development tool ecosystem[**?**]. Such tooling can decrease the development time a lot, and we assume that it would also lead composers to write interactive scores more efficiently.

7. CONCLUSION

- plug-in architecture
- open-source
- soon : audio and midi support (midi can be done today with osc wrappers but a piano-roll or sheet music view would be nice)
- computation graph may be useful to plug the input of a box to an output without using the tree
- other features : spatial, etc.
- embeddable
- problématique de déboguage, analyse statique

- dire qu'on veut utiliser des retours sur les utilisations les plus courante de JS par les gens pour en faire des éléments prédéfinis de l'interface graphique.

Acknowledgments

This work was funded by Blue Yeti and ANR.

8. REFERENCES

[1] P. Ackermann, "Direct manipulation of temporal structures in a multimedia application framework," in *Proceedings of the second ACM international conference on Multimedia*. ACM, 1994, pp. 51–58.

[2] J. Song, G. Ramalingam, R. Miller, and B.-K. Yi, "Interactive authoring of multimedia documents in a constraint-based authoring system," *Multimedia Systems*, vol. 7, no. 5, pp. 424–437, 1999.

[3] J.-M. Celerier, P. Baltazar, C. Bossut, N. Vuaille, J.-M. Couturier *et al.*, "OSSIA: Towards a unified interface for scoring time and interaction," in *Proceedings of the 2015 TENOR Conference*, 2015.

[4] P. Baltazar, T. de la Hogue, and M. Desainte-Catherine, "i-score, an Interactive Sequencer for the Intermedia Arts," pp. 1826–1829, 2014.

[5] C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Commun. ACM*, vol. 9, no. 5, pp. 366–371, May 1966.

[6] H. D. Mills, "Mathematical foundations for structured programming," 1972.