# A POLY-TEMPORAL PROGRAMMING ENVIRONMENT FOR LIVE SHOWS AND INTERACTIVE INSTALLATIONS

**Martin Fouilleul**
Sorbonne Université - STMS Ircam
martin.fouilleul@ircam.fr

**Jean-Louis Giavitto**
CNRS - STMS Ircam
jean-louis.giavitto@ircam.fr

**Jean Bresson**
Ableton, Berlin - STMS Ircam
jean.bresson@ircam.fr

## ABSTRACT

In this paper we present *Quadrant*, a prototype programming environment for designing and performing temporal scenarios. Such scenarios can be used to trigger cues and control various parameters of live shows and interactive installations, such as audio and video playback, lights, or mechatronics.

Quadrant features a structure editor operating on a syntax tree that intermingles textual tokens and graphical user interface elements. This allows specifying scenarios algorithmically using a domain specific language, while expressing continuous time transformations with graphical curves.

Quadrant uses an imperative synchronous language to express concurrent poly-temporal scenarios. Scenarios are compiled on-the-fly into a bytecode that is run by a virtual machine. A temporal scheduler organizes the execution of concurrent parts of that bytecode along multiple abstract time axes, mapping abstract dates and delays of the program onto real time using a differential equation solver.

The virtual machine feeds back execution information to the structure editor, which reflects that information by highlighting executed statements or displaying progress wheels and status icons directly in the code. This allows an operator to easily monitor and pace the progression of the scenario.

## 1. INTRODUCTION

Timing and interactions are two critical aspects of almost any live show. Likewise, a large number of art installations are experienced within a specific temporality, and/or feature complex interactions. These two broad categories of artworks also increasingly involve technological artifacts and processes, as artists take advantage of them for creative purposes. Generative audio, shaders, video mapping, LED arrays, various sensors and robotic devices, are a few of the elements that can now participate in a rich network of temporal interactions with the performers and the audience.

In this context, technical designers or tech-savvy artists need tools to plan and control the *temporal scenario* of the show or installation. Such tools face somewhat opposed requirements: they must offer tight control over the scenario's timing, while remaining flexible and open enough to allow creative outcomes. They must provide a useful view of the show at multiple levels, e.g. from the broad strokes of the plot down to the details of a fader's automation. They must be ergonomic and easily controllable in live, while allowing the design and automation of complex processes. Tradeoffs are to be made, and can be made in a variety of ways. Furthermore, there is no one true solution that will work for all artists and every show. The design space is thus quite large and calls for exploration.

A number of tools have been proposed that enable the authoring and execution of temporal scenarios for various types of media. Cuelist softwares such as QLab [1], Medialon [2] or Smode [3] use nested lists of cues with ad-hoc timing behaviours, and/or timelines reminiscent of audio sequencers. Iannix [4] uses a spatial metaphor with 3D objects and trajectories to organize and schedule actions. Ossia Score [5] adopts a flow graph model which provides some degree of abstraction and programmability, while still conveying a spatial metaphor of time. Formula [6], Antescofo [7] or ChucK [8] tackle the problem by defining domain specific programming languages with temporal semantics. Gibber [9] relies on a general purpose scripting language (Javascript) and notation conventions to build live-coded multimedia performances.

In this work, we present a prototype environment called Quadrant, that aims at bridging the gap between a programming language approach and a more user interface focused point of view [1]. Quadrant consists of three components: a structure editor, a non-textual domain-specific language with a bytecode compiler, and a virtual machine. Bringing the full expressive power of a programming language to technical users affords them both precise control and unbounded extensibility. In a standard textual approach, this may come at the cost of learnability, efficiency, ergonomics and immediate feedback. However, the tight integration of Quadrant's components, and the structured representation of temporal scenarios they share, enables important user experience improvements.

## 2. QUADRANT STRUCTURE EDITOR

Quadrant features a structure editor named *Qed* to encode temporal scenarios. Structure editors let users view and edit structured data, using specific knowledge of the underlying format. This is to contrast with text editors, which

[1] A short video overview of Quadrant is available here: https://youtu.be/_wGPEDwp1AA.

operate on mostly unstructured streams of bytes, even when those streams are meant to encode some structure, such as a program. Although structure editors mostly gained wide adoption *outside* computer science academia [2], there is a long history of research into structural code editing. It ranges from early syntax-directed editors, such as MENTOR [10], the Cornell Program Synthesizer [11], or editor generators explored by the Gandalf Project [12], to more recent research rooted in type theory and functional programming such as Hazel [13] or Tylr [14], non-textual language generators like JetBrain's MPS [15], or attempts at designing general purpose structured formats such as Dion [16] or Infra [17].

### 2.1 Structured Data

The goal of Qed is to provide more ergonomic input and visualization models for temporal features of the language, as well as useful feedback about the execution of the program. However the editor should in some cases preserve the *feel* of text editing. First, it is still an appropriate *local* model for a lot of interactions (e.g. typing exact numeric values, simple arithmetic expressions, and obviously, text strings). More fundamentally, the editor should ease exploration of the user's problem space and incremental progress towards a solution. This sometimes implies relaxing structural requirements and allowing the user to navigate through error states. This departs from the insistence of most structure editors on making syntax or type errors impossible, which in our opinion is the main contributor to their perceived "stiffness" or lack of flexibility.

Qed is thus less structured than most structure editors, and defers checking some of the structural constraint to later stages of the compiler pipeline. The editor operates on a tree structure composed of *cells*. Cells can be simple tokens such as numbers or textual identifiers, or lists of other cells.

### 2.2 Navigation and Edition

The editor maintains a cursor which corresponds to a position inside the tree. The cursor can be described by a triple $(p, r, o)$, where $p$ is the parent cell under which the cursor lies, $r$ is the children of $p$ which is immediately on the right of the cursor (which can be null if the cursor is at the end of $p$'s children list), and $o$ is a text offset into the textual data of $p$, if any.

The cursor can be moved backwards or forwards in depth-first traversal order using left and right arrow keys, It can also be moved to the position that is *visually* upwards or downwards, using up and down arrow keys. The editor displays hints to help visualize the position in the tree: the parent cell is rendered against a light gray background, and the cells left and right to the cursor are indicated respectively by blue and green underlines.

A secondary cursor called the *point* is used to select parts of the tree. The point normally follows the movement of the cursor, unless the shift key is pressed, in which case it stays in place. To infer a selection from the cursor and point, we first determine their closest common ancestor $\mathscr{P}$. The selected range is then the minimal forest consisting of consecutive children of $\mathscr{P}$ which contains the point and the cursor. This allows growing and shrinking the selection to the next meaningful syntactic boundary as the cursor moves towards or away from the point.

Selections can be deleted, copied and pasted with standard shortcuts. In addition, when the cursor is not inside a string literal cell, the editor recognizes some keystrokes as special editing shortcuts to allow editing the structure of the tree: for instance, pressing the left parenthesis `(` key creates a simple list cell, while the shortcut `Cmd+(` encloses the cell directly right to the cursor into a new list cell.

Otherwise, entering text replaces the current textual selection by the input characters, much as in a standard text editor. Once the input characters have been inserted, the cell's text is tokenized, which may modify the current cell or split it into several new cells.

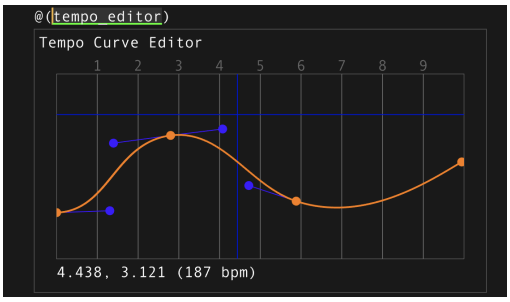


Figure 1. Quadrant Tempo Curve Editor

### 2.3 Tempo Curve Editor

Since the editor is aware of language structure, it can recognize certain syntactical constructs, and attach custom graphical user interface elements to them. This is the case with the tempo curve editor (Figure 1), which allows specifying the tempo of a block of code with Bézier curves, whose control points can be adjusted with the mouse. This offers both an easier input model and a better visualization of continuous curves than a purely textual construct.

The user can zoom and scroll in an infinite grid with adaptive markings. Clicking on the curve splits the Bézier curve under the mouse. Alt-clicking on an endpoint deletes it and merges the two pieces on each side. The user can drag control points with the mouse to shape the curve as needed. A dark blue crosshair and textual coordinates help precisely control the position of the mouse pointer. Clicking an endpoint and hitting `Ctrl+C` toggles the tempo continuity constraint at this point. When the constraint is disabled, the last and first points of two consecutive Bézier curves can be dragged up and down independently.

## 3. TEMPORAL MODEL

We briefly discuss the main aspects of Quadrant's temporal model here. For a more detailed explanation, we refer the

[2] Word processors, cell sheets or 3D modelling softwares are good examples of widely used structure editors. Meanwhile in 2022, computer science papers, including this one, are still being written in LateX.

reader to a previous work [18] on poly-temporal scheduling with parametric tempo curves.

### 3.1 Symbolic timeframes

A musical score ascribes temporality to musical events using symbolic dates and durations (e.g. beats and notes values). Symbolic time is mapped to performance time by the interpreter, following tempo indications, cultural conventions, and interpretative choices. In the realm of computer music environments, several authors have proposed extending this mapping recursively, creating a hierarchy of symbolic timeframes, where a timeframe is related to its parent by a time transformation [6, 19, 20]. Time transformations can be represented and applied by a variety of techniques: precomputed time maps [19, 21], tempo and time shift maps [22], beta functions [23], piecewise tempo curves built on tweening functions [24].

Mapping a date from one timeframe to another using a tempo curve implies solving a differential equation [18]. Instead of relying on predefined functions with known integrals, Quadrant uses a numerical solver to integrate parametric tempo curves. In particular, we chose to construct tempo functions from Bézier curve pieces, which are more versatile than standard tweeners and allow easy tweaking of control points by a user through a graphical interface.

### 3.2 Scheduling

Quadrant's poly-temporal model relies on concurrent tasks (implemented as stackful coroutines, or *fibers*) managed by a cooperative scheduler. Each task represents a sequence of interleaved computations and delays happening in a given timeframe.

Computations are predictably ordered and are considered to happen instantaneously with respect to symbolic time. This makes Quadrant's model similar to that of synchronous languages [25], with a few differences that we detail below.

Strictly speaking, most synchronous languages don't have an inbuilt notion of time, and can only react to signals. This does not pose theoretical difficulties but does make some scenarios cumbersome, since one must rely on introducing and counting external "clock" signals. This downside is discussed in [26], which also proposes extending the host context of Esterel to allow a program to schedule its own wakeup time when returning from its `step` function. We use a somewhat similar approach in Quadrant, where tasks can pause for a requested amount of symbolic time.

We allow temporarily removing some task from the synchronous scheduling mechanism to have them executed in a background task pool. This permits graceful handling of blocking or asynchronous operations, such as input/output, without stalling the scheduler.

Finally, while most synchronous languages are concerned with providing hard real-time guarantees, we are mostly interested in providing a predictable yet flexible concurrency model, and only consider soft real-time goals on a best-effort basis.

## 4. QSCORE LANGUAGE

Quadrant provides a custom language named QScore to encode temporal scenarios. As discussed in section 2, it is a non-textual language: although its source representation is mostly *displayed* as text, it is in fact a tree of cells that holds tokenized data or user interface widgets.

Since the tree structure is rendered explicit by the editor, in the form of indentations and parenthesis and through the use of s-expressions, the appearance of the source is quite reminiscent of the Lisp programming language. This is however the end of the similarity, since QScore has very different characteristics than typical Lisp dialects. Indeed, QScore is an imperative, statically typed language with (mostly) unmanaged memory. It has built-in cooperative concurrency and temporal primitives based on tasks. It is compiled to a bytecode run by a virtual machine. Since the language is statically typed, values are unboxed and memory layouts are known ahead of time, which means that the virtual machine can be fairly lightweight.

### 4.1 QScore Basic Constructs

We give here a brief overview of the language. Some more advanced features are shown in a short video here: `https://youtu.be/AmO9hczGkYU`.

#### *4.1.1 Variables and Expressions*

The following form declares a variable `name` with type `typeSpec` is the current scope, and initializes it with the expression `initExpr`:

```
(var name typeSpec initExpr)
```

A variable can be assigned a new value of its type using the `set` form:

```
(set name val)
```

Common operators can be used in prefix notation to compute numeric or boolean expressions, such as

```
(and foo (< bar (* 3.2 baz)))
```

#### *4.1.2 Named Types*

A named type can be defined by associating a name to a type specification, using the form:

```
(type typeName typeSpec)
```

where `typeName` is an identifier and `typeSpec` is a type specification. QScore predefines a number of primitive named types for signed and unsigned sized integers, floating point numbers, booleans and a `void` type.

#### *4.1.3 Arrays and Slices*

An array is fixed-size container of contiguous elements of the same type. An array type is specified using this form:

```
(array count typeSpec)
```

A slice is a reference to a contiguous range of elements of an array. Its number of elements need not be known at compile time. Its type is specified using this form:

```
(slice typeSpec)
```

#### *4.1.4 Structures*

A structure is a collection of named fields, each with their own type. A struct type is specified using the form

```
(struct name1 typeSpec1
        name2 typeSpec2
        ...)
```

#### *4.1.5 Pointers*

A pointer is an address of a value of a given type. A pointer type is specified using the form:

```
(ptr typeSpec)
```

The `ref` form takes the address of an addressable operand. The `unref` form allows accessing the underlying value:

```
(ref addressableOperand)
(unref pointer)
```

#### *4.1.6 Control Flow*

QScore has the usual basic control flow constructs like conditionals and loops, and lexical scoping:

```
(if condition
    branchIfTrue
    branchIfFalse)

(for initStatement
     conditionExpression
     iterationExpression
   body)

(while condition
   body)

(do
   body)
```

#### *4.1.7 Procedures*

Procedures can be defined using the `def` form:

```
(def procName (param1 typeSpec1
               param2 typeSpec2
               ...
               -> returnTypeSpec)
   body)
```

A procedure returns values using the `return` form:

```
(return val)
```

Procedure calls use prefix notation:

```
(procName arg1 arg2 ...)
```

### 4.2 Temporal Features

#### *4.2.1 Pause*

A task can request to be paused for a given duration and yield to the scheduler using the `pause` instruction:

```
(pause duration)
```

where `duration` is a numeric value specifying the duration of the pause in the timeframe of the current task.

#### *4.2.2 Standby*

The `(standby)` instruction suspends the execution of the current task until it is resumed by a trigger action sent from the editor.

#### *4.2.3 Flow and Futures*

The `flow` form launches a new task to execute its body, and yields to the scheduler.

```
(flow
   body)
```

A `flow` block can refer to variables from its outer scopes. In this case, it captures those variables, and the activation frames in which these variables live are kept valid at least until the block ends or returns.

The `@(tempo_editor)` attribute can be used right after the `flow` keyword to attach a tempo curve editor to the block (see subsection 2.3). The tempo curve is then used to map the timeframe of the task running the `flow` body to and from its parent task's timeframe.

A `flow` block may contain `return` statements, which must all be of the same type. Otherwise it is considered to return `void` after the last statement. In the calling task, the `flow` form evaluates to a *future*, which is a typed handle to the new task concurrently executing the `flow` body. The type specification of a future is

```
(future typeSpec)
```

where `typeSpec` is the specification of the type returned by the `flow` block.

The `wait` form suspends the current task until a given `flow` block returns. The task is then resumed and the `wait` form evaluates to the value returned by that block.

```
(wait someFuture)
```

The `timeout` form suspends the current task until a given `flow` block returns, or a given local timeout expires, whichever is the earliest. It evaluates to a boolean value which indicates if the `flow` block has returned.

```
(timeout someFuture maxDuration)
```

`wait` and `timeout` forms can use the `@(recursive)` attribute to recursively wait for a `flow` block and all the `flow` blocks it launched to have returned.

Tasks are reference-counted to keep the associated data (in particular, the returned value) alive for subsequent waits and timeouts. It is the responsibility of the programmer to manifest their intent in this regard by using the `fdup` and `fdrop` forms, which respectively duplicate a future and increment the reference count of the underlying task, or drop a future, which decrements this reference count.

```
(fdup someFuture)
(fdrop someFuture)
```

### 4.3 Compiler pipeline

The cell tree is processed by several pipelined stages before it can be executed by the virtual machine (Figure 2).

It is first traversed by a parser to produce an abstract syntax tree. A checker then traverses the syntax tree to create a typed syntax tree and symbol tables. The parser and the checker also produce an error log, with each error attached to the range of cells from which it originates. This error

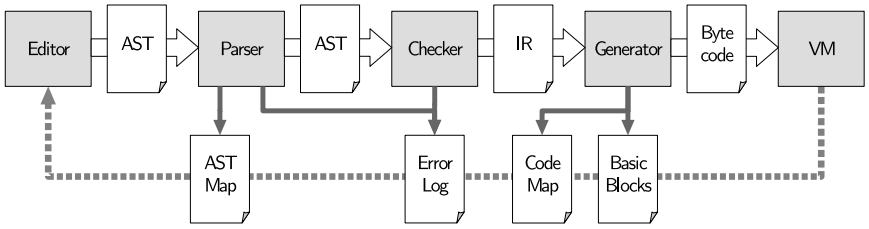


Figure 2. Quadrant Pipeline

log is used by the editor to draw error underlines and display a pop-up panel with error messages when the cursor is on a faulty cell.

When the program is valid, a generator uses the checker output to generate a serialized representation which can be loaded and executed by the virtual machine. This representation contains directives to setup the address space of the program, initialize static data, and load foreign libraries and symbols, as well as the bytecode corresponding to each procedure of the program.

The compiler pipeline generates several debug tables:

- An AST map, which maps cells to AST nodes. This is used by the editor to query syntactic properties of cells, in order to perform auto-layout and syntax highlighting.
- A code map, which contains mappings between cells and bytecode offsets and vice-versa. This table is used by the editor to translate source locations to and from bytecode locations when communicating with the virtual machine.
- A block map, which maps bytecode offsets to basic execution blocks. Basic execution blocks are used by the editor to highlight portions of the source when they are executed by the virtual machine.

## 5. QUADRANT VIRTUAL MACHINE

The virtual machine of Quadrant is a fairly simple stack machine whose design draws some inspiration from the Quake III Arena VM by Id Software [3] .

### 5.1 Instructions Encoding

Code is segregated from program data. Instructions consists of a one byte opcode, followed by zero to three immediate operands. The size of each operand is encoded in the opcode and can be 1 to 8 bytes. Along with standard operations such as loads and stores, arithmetics, comparisons, logic, conversions, and jumps, the instruction set includes specialized opcodes to manage tasks and control time flow.

### 5.2 Address Space Layout

Tasks share the same linear address space, which is reserved through the operating system's virtual memory API at load-time and committed as needed on a page-by-page basis. Data loads and stores are confined to this fixed, contiguous region of memory, which is divided in several sections:

- `rodata`: this section is initialized at VM load-time with the static program data generated by the compiler pipeline, such as string literals and tempo curve descriptors.
- `bss`: this section is initialized to zero at VM load-time, and holds the program's global variables.
- `stack pool`: this uninitialized section is used to allocate fixed-size stacks for the tasks, using a pool allocator.
- `heap`: this uninitialized section is used to allocate objects of different sizes and lifetimes using a general purpose allocator.

### 5.3 Task Structure

The VM keeps track of tasks with a list of `vm_task` structures allocated from a dedicated pool. A generational index maps `future` values to `vm_task` structures. A `vm_task` structure holds the task's scheduling data and registers. It also has a slot to hold the return value of the task. If the task returns a structure, this slot holds a pointer to the return value, which is allocated on the heap.

In addition to the task structure, the VM also allocates a fixed size block from the `stack pool` for each task. This block is further split into two stacks: an operand stack, and a control stack.

For each task the VM maintains two reference counts: the number of `future` objects used to reference that task, as counted by `fdup` and `fdrop` forms, and the number of active captures of that task's control stack. When the number of captures drops to zero, the task's stacks can be recycled by the stack pool. When the active `future` count drops to zero, the memory allocated to hold the return value (if any) can be released to the heap. When both counts drop to zero, the VM can recycle the task structure.

### 5.4 Registers and Stacks

Each task has the following set of registers:

- `ip` (instruction pointer): this register points to the next instruction to execute.
- `osp` (operand stack pointer): this register points to the top of the operand stack.

[3] `https://github.com/id-Software/Quake-III-Arena/blob/master/code/qcommon/vm_interpreted.c`

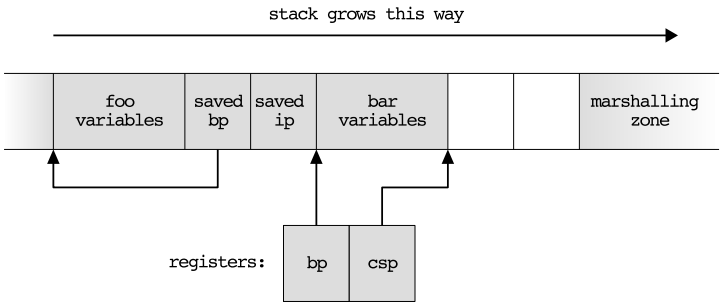


Figure 3. Control Stack Layout (here current procedure `bar` has been called by `foo`)

- `csp` (control stack pointer): this register points to the top of the control stack.
- `bp` (frame base pointer): this register points to the base of the current activation frame.
- `sr` (status register): this register holds status flags used by comparison and conditional jumps.

The operand stack consists of 8-byte aligned operands. Instructions that push or pop values of smaller size respectively zero-extend or mask those values.

The control stack consists of activation frames that contain the local variables of the task's active procedures (Figure 3). After the local variables, two 8-byte slots are reserved to store the `bp` and `ip` registers when calling a procedure. The *marshalling zone* after these slots corresponds to the location of the local variables in the next activation frame, and is used for passing arguments to procedures and new tasks.

### 5.5 Calling conventions

#### *5.5.1 Regular Procedure Calls*

When calling a procedure, the generator outputs opcodes to evaluate arguments on the operand stack, and then move them to the marshalling zone. If the procedure returns a structure, a return pointer is generated and passed as a hidden first argument. It then outputs a `call` opcode. This instruction copies the current frame base pointer and the instruction pointer to their respective slots at the end of the frame, then jumps to the address of the procedure. The procedure's code starts with a `enter` opcode that adjusts the `bp` and `csp` registers to the new activation frame.

#### *5.5.2 Task Calls*

A new task is created either for a `flow` block or for a regular procedure whose frame is captured by `flow` blocks. If necessary, the generator inserts an opcode to allocate memory for the return value prior to the call. The arguments are evaluated and collected to the marshalling zone the same way as a normal call. The `task` opcode then creates a child task and copies the arguments from the marshalling zone to the control stack of the new task. It then creates a `future` value for the new task and puts it on the caller's operand stack. The caller then yields to the scheduler to pass execution to the new task.

#### *5.5.3 Returns*

Upon return, the `csp` register is reset to the `bp` register. If the `csp` register then points to the base of the control stack, the VM pops the return value from the operand stack, stores it in the task's structure return field, and terminates the task. Otherwise, the previous `bp` and `ip` registers are popped from the control stack, which will return to the caller with the return value still on top of the operand stack.

#### *5.5.4 Foreign Calls*

The generator associates an index to each foreign procedure, and generates a listing of foreign dependencies and symbols, including type information describing each procedure's interface. At load time, the VM loads foreign libraries and symbols, and prepares the data structures used by the underlying FFI library (`libffi` [4]) into an array.

A foreign call start by the same argument evaluation sequence as regular calls. The `ffi_call` opcode then populates an argument buffer with pointers to the arguments inside the marshalling zone, obtains the FFI data using the procedure index, and uses the FFI API to make the call.

## 6. EXECUTION MONITORING AND PACING

The execution of a Quadrant program can be monitored and paced from within the editor (Figure 4):

- The editor highlights blocks of code in green as they are being executed.
- The editor shows a progress wheel when a task is paused. The proportion of the green arc shows the proportion of time elapsed with respect to the pause duration.
- The editor displays a spinning wheel composed of two green arcs when a task is waiting on a future.
- The editor shows a standby icon composed of two wavering green rectangles when a task is on standby.
- The editor can send triggers to resume a task suspended by a `standby` instruction

This is achieved by running the virtual machine and the editor in separate threads and have them communicate via message passing using a pair of wait-free ring buffers.

[4] `https://github.com/libffi/libffi`

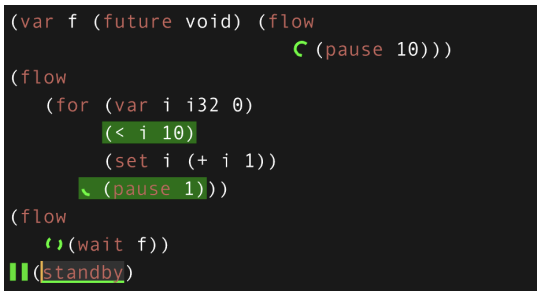


Figure 4. Quadrant Execution Monitoring

### 6.1 Execution Blocks

An execution blocks is a contiguous, uninterrupted block of bytecode (i.e. containing no jumps and yields) that maps to a contiguous range of cells. The generator produces a table pairing the start offsets of execution blocks to the range of cells they originate from.

When executing a jump or a yielding instruction, the VM sends a `trace` message to the editor, containing the target offset of the jump. The editor then uses the blocks table to map it to a range of executed cells, and displays a flashing green background behind those cells.

Execution may fall back from one execution block to another without a jump or yielding instruction. This is e.g. the case when joining from the `false` branch of an `if` form to the next execution block. It can also happen when the generator reorders instructions compared to how they appear in the source. In these cases, the generator inserts a `trace` opcode at the beginning of the second execution block, which explicitly instructs the VM to emit a `trace` message.

### 6.2 Progress Reports

The scheduler runs a special fiber at a fixed frequency to monitor the progress of the scenario. For each task, the monitoring fiber collects the status of the task, the time remaining, and a call stack consisting of the bytecode offset of all call-sites for that task up to the current `ip`. It then sends a `progress` message to the editor containing that information. The editor then uses the code map to display progress wheels, spinning wheels or standby icons next to all call sites of a suspended task.

### 6.3 Standby Triggers

When the cursor is on a `standby` form, and the user hits the `Ctrl+Space` shortcut, the editor uses the code map to send a `trigger` message with the bytecode offset corresponding to that form to the VM. The VM then resumes every task that is on standby at that particular location. This provides a way to manually pace the progress of the scenario from within the editor.

## 7. CONCLUSION

In this paper, we pointed out the increasing complexity of human and technological temporal interactions in live shows and art installations. After mentionning a several show control softwares and their paradigms, we introduced Quadrant, a new programming environment for designing and performing poly-temporal scenarios (section 1).

We discussed the motivations behind Quadrant's editor Qed, and presented its main features (section 2). We then explained the poly-temporal model of Quadrant (section 3), and gave an overview of QScore, its non-textual language with built-in temporal constructs (section 4). We also presented the architecture of Quadrant's virtual machine (section 5). We finally described how the Quadrant's editor and virtual machine cooperate to enable live monitoring and control of a temporal scenarios' performance (section 6).

### 7.1 Future Work

Quadrant is obviously still in its infancy. Its ability to interact, both with external devices and with a human operator, is thus quite limited. An immediate avenue for improvement is to extend interactivity on these two fronts.

The first one should be quite easily addressed by constituting a standard library of connectivity protocols, which would include e.g. UDP, TCP and WebSocket, as well as messaging protocols such as OSC, MIDI, or DMX. The foreign system of QScore should prove useful in this endeavour, as a way of leveraging existing code.

The user interactivity aspect is a little more challenging. Our plan is to implement breakpoints and stepping features similar to those of debuggers, and to build interactive workflows on top of it. That would include specialized tasks that act like random access cue lists or timelines, and mechanisms to control their execution from the editor. We would also like to consider live-coding. This could be implemented by appending basic execution blocks to the end of the code section, and patching old blocks with jumps to reroute control flow through these blocks at run-time.

Finally, an important problem to tackle is the automatic synchronization of QScore tasks to external processes, such as sequencers, score followers, or other Quadrant instances. Using the connectivity protocols evoked above, Quadrant could receive beat messages and generate tempo curves on the fly to catch up with these synchronization sources. This would in turn potentially open the way for performances using distributed and collaborative scores.

### Acknowledgments

We would like to thank Allen Webster and Ryan Fleury of Dion Systems for sharing their perspective on structured editing and data formats, and being inspiring interlocutors.

## References

[1] *QLab*. [Online]. Available: `https://qlab.app/`.

[2] *Medialon - Powerful Show Control Solutions*. [Online]. Available: `https://medialon.com/`.

[3] *Real-time compositing, media server and XR - SMODE*. [Online]. Available: https://smode.fr/.

[4] T. Coduys and G. Ferry, "Iannix. Aesthetical/symbolic visualisations for hypermedia composition," in *Sound and Music Computing Conference (SMC)*, 2004.

[5] J.-M. Celerier, P. Baltazar, C. Bossut, N. Vuaille, J.-M. Couturier, and M. Desainte-Catherine, "OSSIA: Towards a unified interface for scoring time and interaction," in *TENOR 2015 - First International Conference on Technologies for Music Notation and Representation*, 2015.

[6] D. P. Anderson and R. Kuivila, "A system for computer music performance," *ACM Transactions on Computer Systems*, vol. 8, no. 1, 1990.

[7] J. Echeveste, J.-L. Giavitto, and A. Cont, "A Dynamic Timed-Language for Computer-Human Musical Interaction," INRIA, Tech. Rep., 2013.

[8] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4, 2015.

[9] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer, "Gibber: Abstractions for Creative Multimedia Programming," in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014.

[10] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," Tech. Rep., 1980.

[11] T. Teitelbaum and T. Reps, "The Cornell program synthesizer: A syntax-directed programming environment," *Communications of the ACM*, vol. 24, no. 9, 1981.

[12] A. N. Habermann and D. Notkin, "Gandalf: Software development environments," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, 1986.

[13] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer, "Hazelnut: A Bidirectionally Typed Structure Editor Calculus," *ACM SIGPLAN Notices*, vol. 52, no. 1, 2017.

[14] D. Moon, *Tylr*, hazelgrove, 2022. [Online]. Available: https://github.com/hazelgrove/tylr.

[15] *MPS: The Domain-Specific Language Creator by JetBrains*. [Online]. Available: https://www.jetbrains.com/mps/.

[16] *Dion Systems*. [Online]. Available: https://dion.systems/.

[17] C. Hall, T. Standley, and T. Hollerer, "Infra: Structure All the Way Down," *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017.

[18] M. Fouilleul, J. Bresson, and J.-L. Giavitto, "A Polytemporal Model for Musical Scheduling," in *15th International Symposium on Computer Music Multidisciplinary Research*, 2021.

[19] D. Jaffe, "Ensemble timing in computer music," *Computer Music Journal*, vol. 9, no. 4, 1985.

[20] J.-L. Giavitto, J.-M. Echeveste, A. Cont, and P. Cuvillier, "Time, timelines and temporal scopes in the antescofo DSL v1.0," in *International Computer Music Conference (ICMC)*, ICMA, 2017.

[21] J. Rogers, J. Rockstroh, and P. N. Batstone, "Music-Time and Clock-Time Similarities under Tempo Changes," in *International Computer Music Conference*, 1980.

[22] H. Honing, "From Time to Time: The Representation of Timing and Tempo," *Computer Music Journal*, vol. 25, no. 3, 2001.

[23] J. MacCallum and A. Schmeder, "Timewarp: A graphical tool for the control of polyphonic smoothly varying tempos," *International Computer Music Conference, ICMC 2010*, 2010.

[24] *Curve - AntescofoDoc*. [Online]. Available: https://antescofo-doc.ircam.fr/Reference/compound_curve/.

[25] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[26] R. von Hanxleden, T. Bourke, and A. Girault, "Real-time ticks for synchronous programming," in *2017 Forum on Specification and Design Languages (FDL)*, 2017.