# A Temporal Programming Environment for Live Shows and Art Installations

Martin Fouilleul


## ▶ To cite this version:

Martin Fouilleul. A Temporal Programming Environment for Live Shows and Art Installations. Programming Languages [cs.PL]. Sorbonne Université, 2023. English. NNT : . tel-04051156v1

## HAL Id: tel-04051156
## https://hal.science/tel-04051156v1

Submitted on 15 Mar 2023 (v1), last revised 29 Mar 2023 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Sorbonne Université
Ecole Doctorale Informatique, Télécommunications et Electronique de Paris
Laboratoire STMS - Ircam, équipe Représentations Musicales

# Un Environnement de Programmation Temporelle pour le Spectacle Vivant et les Installations Artistiques

Par Martin Fouilleul

Thèse de doctorat d'informatique

Dirigée par Jean Bresson

Présentée et soutenue publiquement le 12 janvier 2023

Devant un jury composé de:

| | |
|---|---|
| Miller Puckette<br>Professeur, Department of Music - U.C. San Diego | *Rapporteur* |
| Tanguy Risset<br>Professeur, CITI - INSA Lyon | *Rapporteur* |
| Myriam Desainte-Catherine<br>Professeur, LaBRI - Bordeaux INP | *Examinatrice* |
| Jean-Pierre Briot<br>Directeur Recherche, CNRS - Lip6 - Sorbonne Université | *Examinateur* |
| Yann Orlarey<br>Responsable de la recherche, GRAME | *Examinateur* |
| Jean Bresson<br>Directeur de recherche, Ircam & Responsable Produit, Ableton | *Directeur de thèse* |
| Jean-Louis Giavitto<br>Directeur recherche, CNRS - STMS, Ircam, Sorbonne Université | *Encadrant de thèse* |

Sorbonne Université

Ecole Doctorale Informatique, Télécommunications et Electronique de Paris

Laboratoire STMS - Ircam, équipe Représentations Musicales

# A Temporal Programming Environment for Live Shows and Art Installations

By Martin Fouilleul

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Supervised by Jean Bresson

Defended on January 12, 2023

Before a jury composed of:

| | |
|---|---|
| Miller Puckette<br>Professor, Department of Music - U.C. San Diego | *Reviewer* |
| Tanguy Risset<br>Professor, CITI - INSA Lyon | *Reviewer* |
| Myriam Desainte-Catherine<br>Professor, LaBRI - Bordeaux INP | *Examiner* |
| Jean-Pierre Briot<br>Senior researcher, CNRS - Lip6 - Sorbonne Université | *Examiner* |
| Yann Orlarey<br>Head of research, GRAME | *Examiner* |
| Jean Bresson<br>Senior researcher, Ircam & Team product owner, Ableton | *Thesis Supervisor* |
| Jean-Louis Giavitto<br>Senior researcher, CNRS - STMS, Ircam, Sorbonne Université | *Thesis co-supervisor* |

**Abstract.** Temporality is a critical aspect of live shows and art installations. Technical artifacts and processes participate in a rich network of temporal interactions with the human performers and/or the audience. In this context, technicians and artists need tools to plan and control the temporal scenarios of their show or installation.

In this work we present Quadrant, a programming environment for designing and performing temporal scenarios. Such scenarios can be used to drive various technical aspects of live shows and art installations, such as audio and video playback, lights, or mechatronics.

We explore an hybrid approach aimed at bridging the gap between a programming language and a show controller. Our environment features a structure editor operating on a syntax tree that combines textual tokens and user interface widgets. This allows specifying scenarios algorithmically using a domain specific language, while expressing continuous time transformations using 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 cooperative scheduler organizes the execution of concurrent flows of that bytecode along multiple time axes, using abstract dates and delays, much like a score uses symbolic positions and durations (e.g. bars and beats) to describe musical time. Abstract time is ultimately mapped onto wall-clock time through the use of time transformations, specified as tempo curves, for which we provide a formalism in terms of differential equations on symbolic position. Tempo curves can be built from cubic Bézier curves.

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

A diferencia de Newton y de Schopenhauer, su antepasado no creìa en un tiempo uniforme, absoluto. Creìa en un infinitas series de tiempos, en una red creciente y vertiginosa de tiempos divergentes, convergentes y paralelos. Esa trama de tiempos que se aproximan, se bifurcan, se cortan o que secularmente se ignoran, abarca *todas* la posibilidades.

— Jorge Luis Borges, *El jardìn de senderos que se bifurcan*

# Contents

**Acknowledgments** **x**

**Résumé de la thèse en Français** **1**

**Introduction** **12**

**1 Current Approaches and Tools** **16**
1.1 Time Metaphors . . . . . . . . . . . . . . . . . . . . . . . . 16
1.1.1 Examples . . . . . . . . . . . . . . . . . . . . . . . . 18
1.1.2 Abstract Time and Poly-Temporality . . . . . . . . . . 21
1.2 Programming Interfaces . . . . . . . . . . . . . . . . . . . . 22
1.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

**2 Preliminary Work** **30**
2.1 QScript: A Textual Temporal Scripting Language . . . . . . . 30
2.1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . 31
2.1.2 Temporality of Statements . . . . . . . . . . . . . . . . 32
2.1.3 Interfacing with Foreign Code and External Software . 33
2.1.4 Limitations . . . . . . . . . . . . . . . . . . . . . . . 33
2.2 QEd: A Cuelist Editor . . . . . . . . . . . . . . . . . . . . . 35
2.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . 35
2.2.2 Limitations . . . . . . . . . . . . . . . . . . . . . . . 36
2.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**3 Introducing Quadrant** **40**
3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.1.1 Interface and Example Program . . . . . . . . . . . . . 42
3.2 Structure Editor . . . . . . . . . . . . . . . . . . . . . . . . 44
3.2.1 User Interface Cells . . . . . . . . . . . . . . . . . . . 44

3.2.2 Navigation and Selection . . . 46
3.2.3 Edition . . . 48
3.3 Feedback and Monitoring . . . 51
3.3.1 Error Reporting and Placeholder Cells . . . 51
3.3.2 Auto-Layout . . . 52
3.3.3 Code Completion . . . 54
3.3.4 Live Performance Monitoring . . . 56
3.4 Conclusion . . . 56

**4 Temporal Model 59**
4.1 Symbolic Timescales . . . 60
4.2 Time Transformations . . . 61
4.2.1 Differential Equation Formulation . . . 63
4.3 Tempo Curves Integration . . . 64
4.3.1 Integration of Constant and Linear Tempo Curves . . . 64
4.3.2 Parametric tempo curves. . . . 65
4.3.3 Bézier Tempo Curves. . . . 66
4.3.4 Multi-segment curves implementation . . . 71
4.4 Phase Synchronization . . . 72
4.4.1 Catch-Up Curves . . . 75
4.5 Scheduler . . . 76
4.5.1 Scheduler API . . . 78
4.5.2 Scheduler's operation . . . 81

**5 Quadrant's Temporal Language 88**
5.1 Basic Constructs . . . 89
5.1.1 Variables and Expressions . . . 89
5.1.2 Control Flow . . . 89
5.1.3 Named Types . . . 90
5.1.4 Arrays and Slices . . . 90
5.1.5 Structures . . . 91
5.1.6 Pointers . . . 91
5.1.7 Conversions . . . 92
5.1.8 Any Type . . . 92
5.1.9 Procedures . . . 93
5.1.10 Variadic Procedures . . . 93
5.1.11 Polymorphic Procedures . . . 94
5.1.12 Module System . . . 95
5.1.13 Foreign Blocks . . . 96
5.2 Temporal Features . . . 97

5.2.1 Pause . . . 97
5.2.2 Standby . . . 97
5.2.3 Flow and Futures . . . 97
5.2.4 Background Pool . . . 99
5.2.5 Phase Synchronization . . . 99
5.3 Example Program . . . 100

**6 Compiler and Runtime Implementation 103**
6.1 Compiler Pipeline . . . 103
6.1.1 Modules Handling . . . 104
6.1.2 Parser . . . 104
6.1.3 Checker . . . 105
6.1.4 Generator . . . 107
6.2 Virtual Machine . . . 110
6.2.1 Instructions Encoding . . . 110
6.2.2 Program Loading . . . 110
6.2.3 Address Space Layout . . . 111
6.2.4 Task Structure . . . 111
6.2.5 Registers and Stacks . . . 112
6.2.6 Calling Convention . . . 113
6.3 Execution Monitoring and Control . . . 114
6.3.1 Execution Blocks . . . 115
6.3.2 Progress Reports . . . 115
6.3.3 Standby Triggers . . . 116
6.3.4 Updating Tempo Curves . . . 116
6.4 Conclusion . . . 116

**7 Surrounding Infrastructure 118**
7.1 Core Modules . . . 118
7.1.1 Memory . . . 119
7.1.2 OSC . . . 119
7.2 Service Discovery . . . 124
7.2.1 Discovery, publication, and revocation . . . 126
7.2.2 Congestion reduction . . . 127
7.2.3 Expiration . . . 127
7.3 Leader Election . . . 127
7.4 Clock Synchronization . . . 131
7.4.1 Custom Clock Synchronization Protocol . . . 132
7.4.2 Clock Synchronization Simulator . . . 133
7.5 An Artistic Application . . . 137

7.5.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . 137
7.5.2 Gestures . . . . . . . . . . . . . . . . . . . . . . . . . . 139
7.5.3 Latency Alignment . . . . . . . . . . . . . . . . . . . . . 140
7.5.4 Productions . . . . . . . . . . . . . . . . . . . . . . . . 141

**8 Conclusion 142**
8.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
8.2 Limitations and Perspectives . . . . . . . . . . . . . . . . . . 144
8.3 Closing Thoughts . . . . . . . . . . . . . . . . . . . . . . . . 148

**References 150**

# Acknowledgments

Most grateful thanks to my supervisors, Jean Bresson and Jean-Louis Giavitto, for their trust, their advice and their support.

Warm thanks and thoughts to my fellow PhD students, Constance, Yann, Paul, Valérian, Clément, Victor, Claire, Nadia, for the solidarity, intellectual emulation, and fun times.

I'd like to thank and give credit to Allen Webster and Ryan Fleury of Dion Systems for sharing their perspective on structured editing and data formats, and being inspiring interlocutors.

I also want to extend heartfelt thanks and gratitude to the fine folks at the Handmade Network and Handmade Seattle, in particular Abner Coimbre, Phil Homan, Ben Visness, Colin Davidson and Asaf Gartner. This community has revived and kept my joy of programming alive and strong and has been a strong source of motivation all these years.

My deepest gratitude and love goes to my parents, my sister, and to Arusyak, who wholeheartedly supported me during this PhD.

# Résumé de la thèse en Français

## Introduction

Le temps est un aspect fondamental de tout spectacle vivant, et de bon nombre d'installations artistiques. Ces œuvres d'art font un usage créatif du temps pour mettre en forme leur dramaturgie, dérouler leur action, créer des contrastes et des climax. Pour une grande partie, l'art de l'interprète consiste à jouer avec le temps : certaines actions doivent se produire à des moments spécifiques, dans un certain ordre, à une vitesse donnée, ou doivent entretenir des relations temporelles complexes avec d'autres actions et évènements. La *perception* du temps est également de la plus haute importance, dans la mesure où c'est elle qui permet de construire de l'anticipation, de produire de la surprise, de créer des tensions et des résolutions.

Acteurs, musiciens, danseurs, comédiens, etc., interprètent des processus temporels abstraits décrits sous la forme de partitions ou de scénarios, pour produire une action en temps réel, ou bien construisent directement la progression temporelle du spectacle au travers de l'improvisation. Pour ce faire, ils doivent continuellement construire un consensus autour d'une notion commune du temps : en d'autres termes, ils doivent se synchroniser. Ce consensus est sujet à un ajustement constant, et doit préserver leur liberté d'interprétation. Par example, certaines parties de l'action peuvent diverger et s'écouler indépendamment, avant de se rejoindre à un point ultérieur, dont la date en temps réel n'est pas décidable *a priori*. Loin d'être réductible à une ligne temporelle déterminée d'avance, les scénarios temporels de spectacles vivants sont composés de multiples *flots temporels* simultanés et interdépendants, auxquels l'interprétations donne une réalisation temporelle concrète.

Bien évidemment, les dispositifs techniques et leurs opérateurs sont également pris dans les même flots temporels, et doivent se synchroniser et

s'adapter en temps réel à la progression de l'action. Ceci est d'autant plus vrai que les œuvres contemporaines impliquent un nombre croissant d'artefacts et de processus technologiques, à mesure que les artistes s'en saisissent pour en tirer parti dans leur démarche créative. Les dispositifs de synthèse audio ou vidéo, les processus de composition générative, le mapping vidéo ou les écrans de LEDs, ou encore divers capteurs et dispositifs robotiques, constituent quelques exemples d'outils qui peuvent désormais s'intégrer dans un riche réseau d'interactions temporelles avec les interprètes et le public.

### Quelques examples dans la musique contemporaine

Parmi les arts vivants, la musique a développé des constructions temporelles particulièrement fines, utilisant diverses représentations symboliques du temps, aussi bien continues que discrètes. Pour cette raison, la musique présente des défis spécifiques et particulièrement intéressant au regard de la composition et de l'interprétation des interactions humains-machines distribuées sur de multiples échelles de temps et flots temporels.

Ces problématiques sont particulièrement prégnantes dans la musique mixte, un genre de la musique contemporaine qui se construit spécifiquement autour de l'interaction entre des musiciens humains et des processus musicaux électroniques ou informatiques. Pour donner quelques examples, on peut citer l'installation interactive *Biotope* de Jean-Luc Hervé, les œuvres de musique mixte de Sasha Blondeau telles que *Urphänomen*, l'orchestre robotique utilisés par Pedro García Velásquez dans *La Selva Virgen*, ou encore l'opéra *Like Flesh* de Sivan Eldar.

Des scénarios temporels complexes impliquant des humains et des machines peuvent également s'inscrire dans un projet de *Gesamtkunstwerk*[1]. C'est par example le cas pour l'opéra *Donnerstag aus Licht* de Karlheinz Stockhausen. La figure 1 montre une photographie d'une répétition de cet opéra par l'ensemble Le Balcon, et met en évidence les différents rôles et postes de travail tenus par les nombreux participants, artistes ou techniciens. L'exécution de cette œuvre implique en effet deux orchestres, des chanteurs, un chœur, des danseurs, un dispositif de projection vidéo, un système de sonorisation, un partie de musique électronique, des jeux de lumière, des décors, des sur-titres, etc. Les interactions et les dépendances temporelles entre ces divers éléments sont elles-mêmes mises en œuvres à travers un certain

[1] Gesamtkunstwerk signifie "œuvre d'art totale". Ce terme caractérise une esthétique consistant à combiner délibérément de nombreuses formes d'art, telles que la musique, le théâtre, la danse, l'architecture, etc., au sein d'une même œuvre d'art.

nombre de canaux, tels que des signaux audio ou visuels, un système d'intercoms, et des protocoles informatiques tels que MIDI ou OSC. Les flèches indiquent ces canaux ainsi que le sens des dépendances temporelles : par exemple, le chef d'orchestre donne la battue à l'orchestre et aux chanteurs, tout en vérifiant son propre tempo par rapport à certaines séquences vidéo. La musique est à son tour une source de synchronisation pour le *topeur*, la personne en charge de suivre la partition et de donner les 'tops' pour signaler différents évènements importants à l'équipe technique, en particulier concernant les lumières, le son, ou la vidéo. Il existe évidement des boucles de rétro-action implicites dans ce schéma, par example entre le chef et la vidéo, bien que les séquences vidéo ne prescrivent qu'un tempo approximatif, tandis que le début des séquences vidéo est lui dicté de manière plus précise par la musique (soit par l'intermédiaire du topeur, soit par l'utilisation d'une "partition vidéo" jouée par un pianiste sur un clavier MIDI).

Ces quelques examples illustrent l'utilisation de l'interaction humains-machines dans le spectacle vivant. Cependant, le degré auquel les artefacts et processus technologiques peuvent être orchestrés pour jouer aux cotés d'interprètes humains dans des scénarios temporels dynamiques détermine largement leur potentiel en tant que dispositif créatif. Ceci présente des défis complexes ayant trait à la composition, l'execution, le contrôle et le retour d'informations en temps réel de ces outils.

Dans ce contexte, les concepteurs techniques ou les artistes ont besoin d'outils pour planifier et contrôler les scénarios temporels de leurs spectacles ou installations. Ces outils font face à des exigences multiples et parfois contradictoires : ils doivent offrir un contrôle précis de la temporalité du scénario, tout en restant suffisamment flexibles et ouverts pour permettre l'exploration créatrice. Ils doivent donner un aperçu utile du spectacle à de multiples échelles, depuis les grands mouvements de l'action jusqu'au détail du mouvement d'un potentiomètre. Ils doivent être ergonomiques et aisément contrôlables en situation de live, tout en permettant l'automatisation et le design de processus complexes. Des compromis doivent être faits, et peuvent l'être de multiples manières. De plus, il n'existe pas de solution unique qui conviendrait à tous les artistes, pour n'importe quel type de spectacle. Le champ de possibilités est donc assez vaste et appelle à l'exploration.

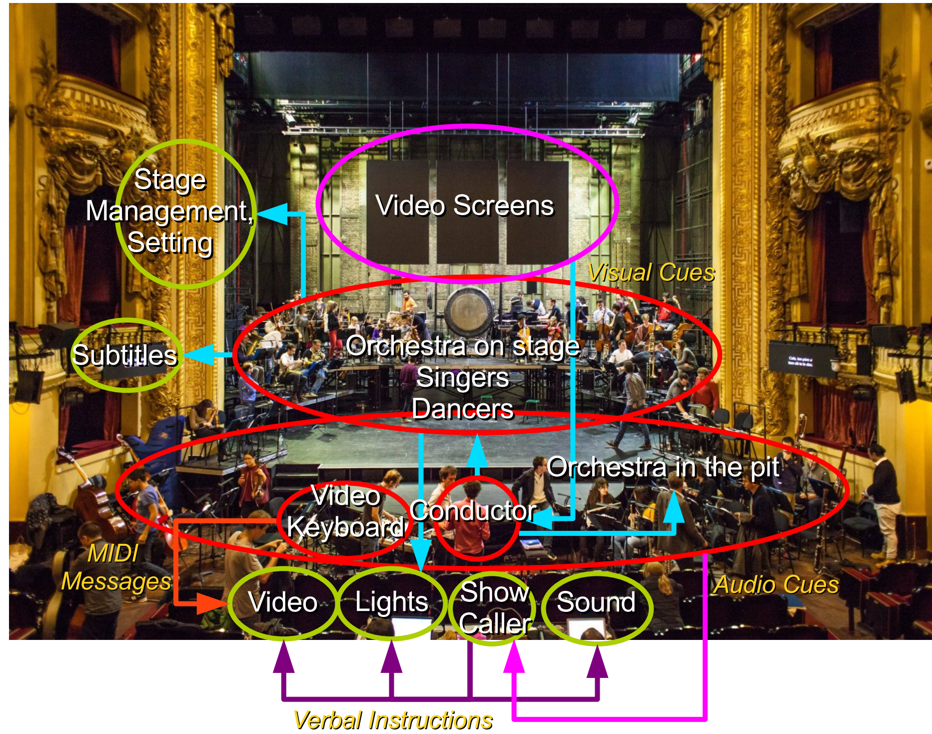



FIGURE 1 : Dépendances temporelles et canaux de communication dans *Donnerstag aus Licht* de K. Stockhausen. Produit par l'ensemble Le Balcon, Novembre 2018. Photographie de Meng Phu.

## Approche développée dans cette thèse

Dans ce travail, nous présentons un environnement de programmation temporelle appelé Quadrant, qui vise à combler le fossé entre une approche de type language de programmation, et un point de vue plus centré sur une interface utilisateur interactive. Nous tentons d'atteindre cet objectif en combinant un éditeur structuré avec un language de programmation dédié, non textuel, et exécuté par une machine virtuelle. Le "code source" est stocké et manipulé sous forme d'arbre, dont les feuilles peuvent être des éléments aussi bien symboliques (par exemple des identifiants textuels ou des nombres) que figuratifs (par exemple des courbes continues). Ceci permet aux utilisateurs d'écrire des scénarios temporels dans un language qui ressemble à un language textuel, tout en intégrant des éléments d'interface graphique. De plus,

l'intégration forte entre les différents composants de Quadrant, et la représentation structurée de scénarios temporels qu'ils partagent, permet d'importantes améliorations de l'expérience utilisateur, tels que la mise en forme automatique, le signalement d'erreurs en ligne, la suggestion de code, ainsi que le retour d'information sur l'exécution du scénario en temps réel.

### Approches et outils existants

Dans le chapitre 1 de la thèse, nous présentons quelques outils existants pour la conduite de spectacle et la composition de séquences multimédia. Nous identifions un certain nombre de métaphores temporelles utilisées par ces outils :

- les *timelines* qui organisent les évènement sur des lignes temporelles statiques ;
- les *cuelists* qui organisent les évènement en listes imbriquées disposant de sémantiques temporelles spécifiques ;
- les métaphores spatiales, qui utilisent des représentations 2D ou 3D, comme des objets et des trajectoires pour représenter des flux temporels ;
- les graphes temporels qui représentent le passage du temps ou des flux de données à travers des graphes d'objets exécutant certaines actions ;
- les approches basées sur des languages de programmation disposant de sémantiques temporelles propres.

Nous présentons également plusieurs interfaces de programmation, et proposons de les replacer sur un axe allant d'une approches symbolique à une approche figurative, distinction que nous jugeons plus fertile que la distinction traditionnelle entre programmation textuelle et programmation visuelle. Nous proposons une identification des forces et faiblesses de chacune de ces approches, et concluons en pointant le potentiel d'une approche hybride, combinant un language temporel symbolique et des éléments d'interface graphiques et de retour d'information figuratifs, rendue possible par un éditeur structuré.

### Travaux préliminaires

Dans le chapitre 2, nous présentons deux prototypes exploratoires ayant servi à guider notre recherche. Le premier est un language de programmation tem-

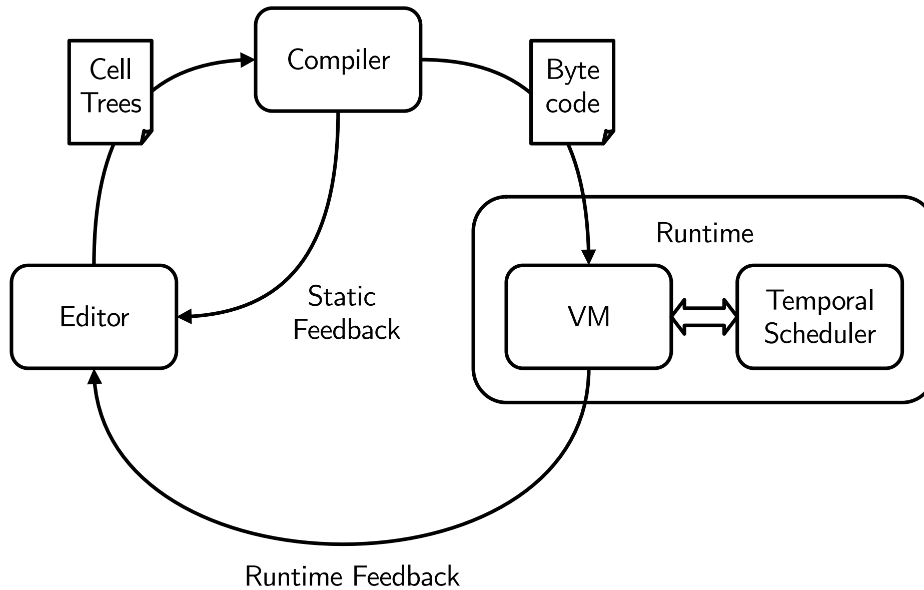



FIGURE 2 : Architecture de Quadrant

porel textuel, tandis que le second est un logiciel de conduite de spectacle basé sur des listes de *cues* hiérarchiques. Nous explorons leur limitations et observons que les idées pour pallier à ces limitations convergent vers une approche hybride commune. D'une part, le besoin d'introduire des éléments d'interface graphique figuratives dans le language textuel conduisent à brouiller les frontières entre le language, son éditeur et son environnement d'exécution. D'autre part, le fait de rendre plus modulaire et re-combinable les blocs élémentaires du logiciel de conduite de spectacle suggèrent un modèle plus programmatique et symbolique.

### Présentation de Quadrant

Au chapitre 3, nous présentons notre environnement de programmation temporelle. Nous définissons les objectifs à l'origine de sa conception, et donnons un aperçu de son architecture (figure 3.1).

Notre approche se fonde sur un format non textuel permettant de représenter des scénarios temporels dans un language de programmation dédié. L'environnement combine :

- Un éditeur structuré qui est utilisé à la fois pour composer des scénarios et contrôler et suivre leur exécution durant le spectacle.
- Un pipeline de compilation produisant un bytecode dédié à partir de la représentation éditable du scénarios.
- Un environnement d'exécution consistant en une machine virtuelle et un ordonnanceur temporel.

Deux chemins de retour d'information existent dans cette architecture. Le premier opère au moment de l'édition et fournit un retour syntactique et sémantique du compilateur vers l'éditeur, permettant d'implémenter les fonctionnalités de l'éditeur telles que la mise en forme automatique, le signalement d'erreurs et la suggestion de code. L'autre opère lors de l'exécution, et fournit des informations sur le déroulement du scénario qui peuvent être visualisées dans l'éditeur.

Nous concluons le chapitre en évoquant un certain nombre d'améliorations possibles de l'interface utilisateur, ayant pour objectif de fournir aux utilisateurs une vue aussi complète que possible du déroulement et de la temporalité de leur scénarios.

### Modèle temporel

Le chapitre 4 décrit le modèle temporel de Quadrant, et l'ordonnanceur temporel qui implémente ce modèle. Notre modèle permet d'ordonnancer des calculs et des actions le long d'axes temporels symboliques (ou référentiels temporels) simultanés organisés de manière hiérarchique. Chaque référentiel temporel est synchronisé à son parent à travers une transformation temporelle, qui permet de convertir les unité de temps du référentiel dans celles de son parent, comme montré dans la figure 4.1. Les transformations temporelles peuvent être spécifiées par des courbes de tempo qui peuvent être construites par morceaux à l'aide de courbes de Bézier cubiques.

Nous discutons des notions de référentiel temporel et de transformations temporelles et de la manière dont elles sont traitées dans un certain nombre de travaux antérieurs. Nous donnons un formalisme des transformations temporelles basées sur des courbes de tempo, en termes d'équations différentielles. Nous considérons des tempi constants et linéaires, que nous résolvons analytiquement. Nous considérons également des tempi décrits par des courbes paramétriques, pour lesquelles nous nous tournons vers des méthodes numériques. Nous détaillons les spécificités de courbes de tempo spé-

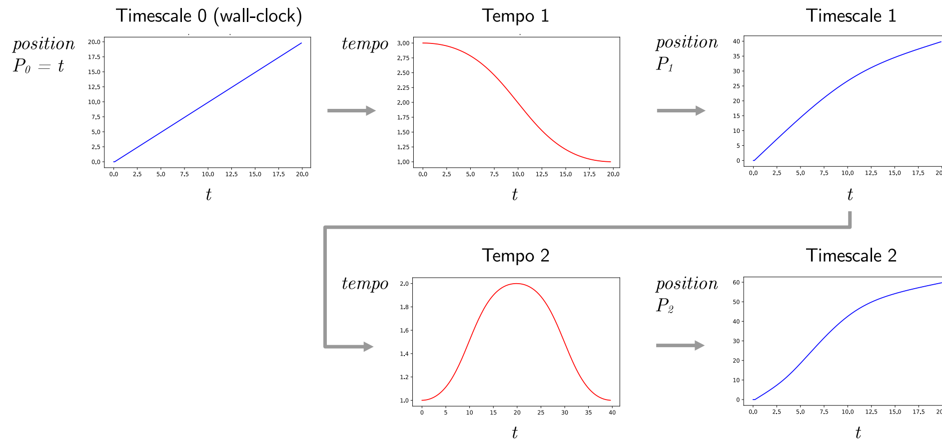



FIGURE 3 : Composition de transformations temporelles utilisant des courbes de tempo.

cifiées par des courbes de Bézier cubiques, et détaillons l'implémentation des courbes de tempo définies par morceau. Nous mentionnons le problème de la synchronisation de phase (et non seulement de vitesse) et expliquons comment nous l'abordons en utilisant des courbes de rattrapage temporel. Nous présentons finalement l'interface de programmation de l'ordonnanceur temporel et présentons les principaux aspects de son implémentation, fondée sur l'utilisation de coroutines.

**Language temporel**

Au chapitre 5 nous présentons le language de programmation temporelle de Quadrant. Il s'agit d'un language non textuel : bien que la représentation des programmes soit principalement affichée comme du texte, elle est en réalité constituée par des arbres pouvant contenir aussi bien des élément textuels que des éléments d'interface graphique.

Comme la structure d'arbre est rendue explicite par l'éditeur, sous la forme d'indentations et de parenthèses, son aspect peut rappeler le language Lisp. La ressemblance s'arrête cependant ici, car Quadrant est en réalité assez différent de la plupart des dialectes de Lisp. En effet, Quadrant est un language impératif, procédural, statiquement typé, avec une gestion de la mémoire principalement manuelle. Il dispose de primitives d'ordonnancement coopératif fondé sur des coroutines. Le language est compilé sous la forme

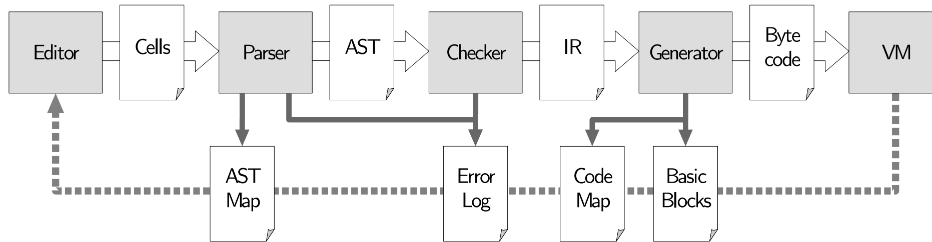



FIGURE 4 : Pipeline de compilation Quadrant

d'un bytecode exécuté par une machine virtuelle. Puisque le language est statiquement typé, les valeurs ne sont pas encapsulées avec leurs informations de type, et l'agencement mémoire des types est connu à la compilation, ce qui permet à la machine virtuelle d'être assez simple et légère.

### Implémentation du compilateur et de l'environnement d'exécution

Au chapitre 6, nous décrivons l'architecture du pipeline de compilation de Quadrant et de son environnement d'exécution (figure 6.1).

Nous détaillons les différents étages du compilateur, et montrons comment certaines constructions du language sont implémentées. Nous décrivons les mécanismes de traçage et de feedback permettant à l'environnement d'exécution de communiquer des informations sur le déroulement du scénario à l'éditeur. Nous montrons également comment l'éditeur utilise ces informations pour afficher des indicateurs permettant à l'utilisateur de suivre l'exécution du scénario.

### Infrastructure environnante

Le chapitre 7 décrit plusieurs composants pouvant former l'embryon d'une infrastructure pour les interactions temporelles distribuées, permettant à Quadrant de piloter et communiquer avec d'autres logiciels et équipements. Nous mentionnons quelques modules formant le début d'une "bibliothèque standard" pour Quadrant et permettant par exemple de communiquer par paquets udp et/ou d'utiliser le protocole OSC. Nous présentons ensuite plusieurs services, qui peuvent aider à l'interopérabilité entre Quadrant et d'autres systèmes. En particulier, nous décrivons un agent de découverte de service, un protocole d'élection, et un protocole de synchronisation d'horloge.

Finalement, nous relatons une collaboration artistique avec le compositeur Pedro García Velásquez, impliquant ces services d'interaction distribuée pour piloter un orchestre de percussions robotiques.

### Conclusion

Nous concluons la thèse en évoquant les limites et perspectives de ce travail :

- **Eléments d'interface utilisateur dédiées :** pour l'instant, le seul élément d'interface utilisateur intégré au language est l'éditeur de courbes de tempo (nous ignorons ici d'autres éléments d'interfaces présents dans l'environnement, tels que le panneau de suggestion de code, ou les roues de progression, car ils ne font pas à proprement parler partie du language lui même mais plutôt de son éditeur). Une avancée évidente serait d'offrir plus d'éléments d'interface graphique dédiés, comme des sélecteurs de couleurs (par exemple pour générer des valeurs RGB à envoyer à des projecteurs LED), des sliders, des aperçus de fichier audio ou d'images, etc. Permettre aux utilisateur eux-mêmes de définir des éléments d'interface graphique à l'intérieur de l'environnement (c'est à dire en utilisant le language de Quadrant lui-même) constitue également une piste importante et ambitieuse.
- **Debugging et profiling intégrés :** la connaissance sémantique partagée entre l'éditeur et l'environnement d'exécution ouvre des pistes qui pourraient être d'avantage exploitées pour fournir des informations de debugging et de profiling en continu dans l'éditeur. Ceci pourrait permettre par exemple l'inspection du contenu de variables ou l'affichage de traces d'exécution, ainsi que l'inspection des temps d'execution ou du nombre d'itérations de chaque bloc de code.
- **Contrôle et déclenchement :** L'environnement pourrait offrir plus de moyens pour contrôler et déclencher l'exécution de code, comme par exemple lancer l'exécution de procédures à la volée depuis l'éditeur, suspendre ou reprendre manuellement l'exécution à des points arbitraire du scénario, ou simplement exécuter les instructions du programme pas à pas ou en avance rapide jusqu'à un point désigné.
- **Développement d'un écosystème :** Quadrant est pensé comme une pièce centrale où est définie et exécutée le scénario du spectacle (de manière analogue au concept de partition électronique centralisée du compositeur José Miguel Fernandez). Ceci signifie que Quadrant doit être capable de communiquer avec un grand nombre de logiciels ou

équipements, tels que des capteurs, moteurs, consoles son ou lumière, synthétiseurs, séquenceurs audio, etc. Ainsi, étendre l'infrastructure d'interaction distribuée décrite au chapitre 7 constitue une part importante du travail nécessaire à faire de Quadrant un outil utile.

- ***Live Coding* :** Dans ce travail, nous n'abordons pas la problématique du *live coding*. Cette omission est délibérée, afin de ne pas disperser nos efforts, mais aussi parce que Quadrant est d'avantage pensé comme un outil permettant de concevoir au préalable des scénarios temporels, et de les exécuter de manière stable pendant un concert, plutôt que comme un instrument d'improvisation créatrice. Cependant, cette pratique peut être très utile comme méthode d'exploration durant la phase de création d'un spectacle, ou pendant les répétitions. L'introduction du *live coding* dans l'architecture et le modèle temporel de Quadrant pose de nombreux défis que nous mentionnons dans la conclusion, et au sujet desquels nous donnons quelques pistes de réflexion.

# Introduction

Temporality is a critical aspect of almost any live show and of a large number of art installations. These artworks make creative use of time to shape their dramaturgy, pace their action, create contrasts and climaxes. By and large, the essence of performance is dealing with time: some events must happen at a specific time, in some order, at some speed, or must maintain complex temporal relationships with other events. *Perception* of time is also of utmost importance, allowing to build anticipation, surprise, tension and resolution.

Performers interpret abstract temporal processes described in the form of a score or scenario to produce a real-time action, or directly build the temporal progression of the show through improvisation. In order to do so, they must continuously form an agreement on some common notion of time: they must synchronize. This agreement is subject to constant adjustment, and must preserve interpretative freedom. For instance, some parts of the action may diverge and flow independently before rejoining at some point, whose absolute date is undecidable *a priori*. Far from being reducible to a predetermined timeline, temporal scenarios of live shows are composed of multiple concurrent and inter-related dynamic *timeflows*, to which the performance gives a concrete, real-time realization.

Of course, technical devices (and their operators) are also caught in the same timeflows, and must synchronize and adapt in real-time to the progression of the action. This is all the more true since live artworks 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.

**A Few Examples in Contemporary Music**

Among performing arts, music has developed particularly fine-grained temporal constructs, using both continuous and discrete symbolic representations of time. As such, it presents specific and interesting challenges with regard to the composition and interpretation of human-machine temporal interactions at multiple scales, and across multiple independent time-flows. This is especially emphasized in *mixed music*, a contemporary music genre that specifically revolves around the interaction between human musicians and computer-generated music. To give a few examples, we refer the reader to the interactive installation *Biotope* by Jean-Luc Hervé, the mixed music works of Sasha Blondeau such as *Urphänomen*, the robotic instruments used by Pedro García Velásquez in *La Selva Virgen*, or to Sivan Eldar's opera *Like Flesh*.

Dynamic temporal scenarios involving humans and machines can also manifest as part of a *Gesamtkunstwerk*[2] project. This is for instance the case of the opera *Donnerstag aus Licht* by Karlheinz Stockhausen. Figure 5 shows a photography of a rehearsal of this opera by the ensemble Le Balcon, highlighting the various positions held by the artists and running crew. This work requires juggling many different medias and actors, including two orchestras, singers, a choir, dancers, a video projection setup and a sound reinforcement system, an electronic music part, light works, stage scenery, subtitles, etc. Interactions and temporal dependencies between these elements are themselves carried over a number of channels, such as visual and audio signals, intercoms, and computer protocols such as MIDI (MIDI Manufacturers Association, 1982) or OSC (Wright, 2002). Arrows indicate these channels and the direction of temporal dependencies: for example, the conductor gives the beat to the orchestras and singers, while cross-checking his timing relative to some of the video sequences. The music in turn is a synchronization source for the show caller, who distributes relevant cues to the lights, sound, and video operators. There are of course implicit feedback loops in this scheme, for example between the conductor and the video, although the video sequences loosely prescribe a tempo, whereas the onset of the sequences are dictated by the music (either through the show caller, or by the use of an explicit video score, played by a pianist on a MIDI keyboard).

[2] A term meaning "total artwork". It characterizes the aesthetic ideals of deliberately combining many art forms such as music, theater, dance, architecture, etc., in one work of art.

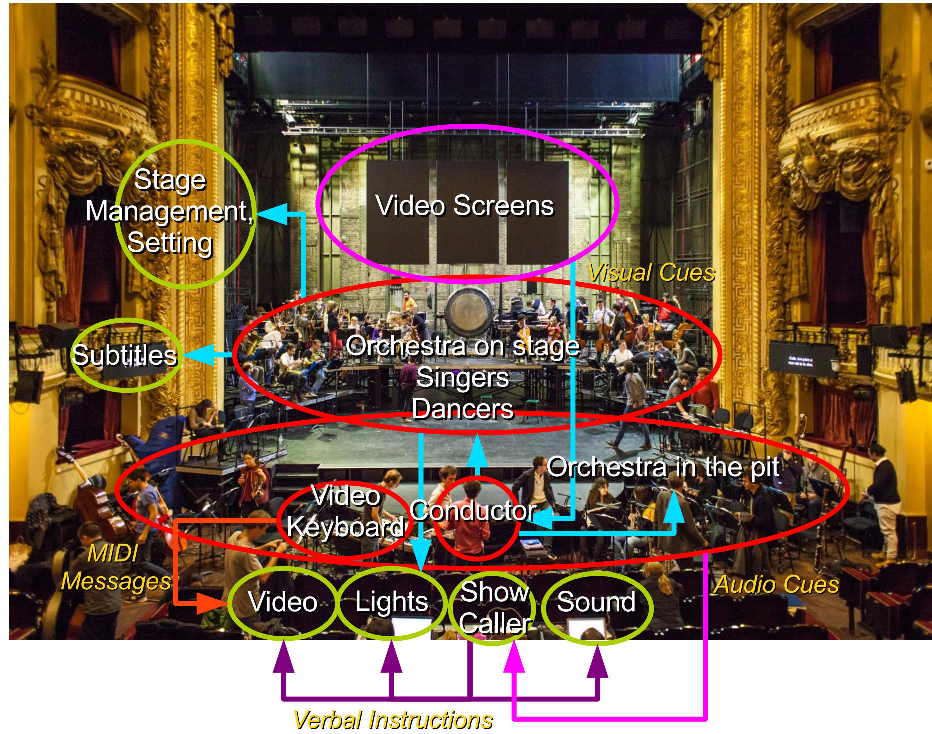



Figure 5: Temporal dependencies and communication channels in *Donnerstag aus Licht*, K. Stockhausen. Produced by the ensemble Le Balcon, November 2018. Photography by Meng Phu.

These few examples illustrate the use of temporal human-machine interactions in artworks. However, the degree to which technological artifacts and processes can be orchestrated to play along human performers within dynamic temporal scenarios largely determines their potential as a creative tool. This presents complex challenges pertaining to authoring and composition, execution, control, and live feedback of these devices.

**Towards a Temporal Programming Environment**

In this context, technical designers or tech-savvy artists need tools to plan and control the temporal scenario of their 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.

In this work, we present a temporal programming environment called Quadrant, that aims at bridging the gap between a programming language approach and a more user-interface focused point of view. It does so by combining a structure editor with a non-textual domain-specific language run by a virtual machine. The "source code" is stored and edited as a tree of tokens, which can be either symbolic or figurative. This allows users to write temporal scenarios in a language that *looks* mostly textual, but can include specialized graphical user interface elements. Furthermore, the tight integration of Quadrant's components, and the structured representation of temporal scenarios they share, enables important user experience improvements, such as auto-formatting, inline error signaling, code completion, and live monitoring of the performance.

The remaining of this work proceeds as follows. We first review existing tools and try to delineate a few common patterns, both in term of temporal model and programming interfaces, in chapter 1. We then discuss our preliminary work on two temporal scenarios authoring paradigms and their limitations, which lead to Quadrant's inception, in chapter 2. We introduce the Quadrant environment and describe its user-facing interface in chapter 3. Next we present the temporal model underlying Quadrant's language and runtime in chapter 4. In chapter 5 we describe the language used to program temporal scenarios. The implementation of the runtime, including visual feedback, is detailed in chapter 6. We finally discuss the small infrastructure surrounding Quadrant and allowing it to interface with the outside world in chapter 7.

# Chapter 1

# Current Approaches and Tools

Software used to specify and control the temporal scenario of a show should obviously present their users with some actionable notion of time. The wide range of complex actions they must be able to express implies that they also need to provide some programming capabilities. Additionally, these tools must cater to an audience with varying levels of programming literacy, including computer music designers and digital artists, as well as sound or lighting engineers. Finally, production constraints (e.g. participating to the creative process in sync with the artists during rehearsals and adapting to frequently changing requirements) impose very fast iteration times and a high level of programming flexibility.

In this chapter we first review a few tools used in the domain of live show control, and attempt to delineate common temporal metaphors (section 1.1). We then propose to analyze programming interfaces along a symbolic/figurative spectrum instead of the often used textual/visual distinction, and we discuss several of those interfaces, their advantages, and their drawbacks (section 1.2).

## 1.1 Time Metaphors

A number of software tools are used or have been proposed in order to enable the authoring and execution of temporal scenarios for various types of media. Several approaches can be identified with respect to the metaphors they present to users. However, most real-world tools use several of these metaphors or blur the lines between them. For this reason, we first give a

tentative and non-exhaustive classification below, and then give examples of tools, and how they fit in this classification.

- Timelines organize events with absolute dates along a common time axis. Timelines are often displayed as a fixed tape with a moving play head, and events are laid out linearly with respect to their associated dates. However, this is only one way of presenting them to users, and in our tentative classification, a chronologically sorted list of events with absolute dates equally falls under this category.

- Cuelists are ordered lists of *cues*, i.e. events pertaining to the technical aspects of the show and synchronized to the live action on stage. A notable way cuelist differ from timelines is that cues don't have a predefined associated time: cues can be triggered manually or in response to some external input (e.g. an MIDI message), individually or in chained sequences. Cuelists can also often be nested to form hierarchies of sequences.

- Some approaches hybridize cuelists and timelines by inserting delays between cues of a cuelist. This allows building timed sequences of cues that resemble timelines. However, dates are relative rather than absolute, and cues can always be triggered directly, independently of the progression of a play head.

- Spatial metaphors can be used to represent time and temporal relationships. They allow positioning cues in some abstract space, which maps to real time through the use of trajectories and spatial relationships.

- Timed graphs can be used to represent the passage of time or messages as edges in a graph of cues. As such they can be used as highly branching timelines. They can also easily express some temporal logic within the scenario, e.g. block the passage of time until some condition is met.

- Programmatic approaches tackle the problem by defining domain specific programming languages with temporal semantics. They mostly treat time as an expandable symbolic ressource, that can be "consumed" at some points of the program. This ressource can be viewed as discrete in the case of reactive approaches where events are consumed to produce a reaction, or as continuous in the case of timed approaches that allow explicitly waiting for some amount of symbolic time. They often dissociate the symbolic time of the scenario from the real time of

the performance, and offer some mechanism to map the former to the latter, as discussed in chapter 4.

### 1.1.1 Examples

We give some examples of tools used in production below, and attempt to map them to time metaphors. Screenshots of some of these tools, illustrating the variety of temporal metaphors, are also shown in Figure 1.1.

#### Digital Audio Workstations

Although not meant for that task, digital audio workstations such as Reaper (Cockos, 2006) or Cubase (Steinberg, n.d.) can sometimes be used as a primitive way of organizing audio clips along a single timeline. With the use of markers, it is also possible to use them to trigger individual cues in a random access fashion.

Live (Ableton, n.d.) offers some more flexibility by providing, a *session view* in addition to the timeline. In this view, each track has a number of slots to which clips can be assigned. Clips can then be triggered from the interface or by an external controller in any order, irrespective of the timeline's play head. Clips in consecutive slots of the same track form a group (meaning a track can have many groups, separated by empty slots). A clip's *follow action* determines what happens to other clips in the same group when the clip finishes playing. Follow actions can start playing the same clip again, start the next or previous clips, play a random clip in the group, etc. Each clip actually has two follow actions, one of which is chosen at random every time the clip finished playing. A slider allows selecting the probability of choosing one clip over the other.

#### Show Controllers

Show Controllers are tools used by sound and lighting engineers to create and run cuelists. They allow launching sound and video samples, control mixing and lighting desks, operate motors for mechatronic stage props, and so on. They are often structured arounds lists of named cues, which also feature visual representations such as progress bars, icons, waveforms, etc. List can have different actions when triggered, like starting their first child, starting all children, or starting a random child, etc. The sequencing of actions inside a cuelist can be determined by chaining them with follow actions and delays.

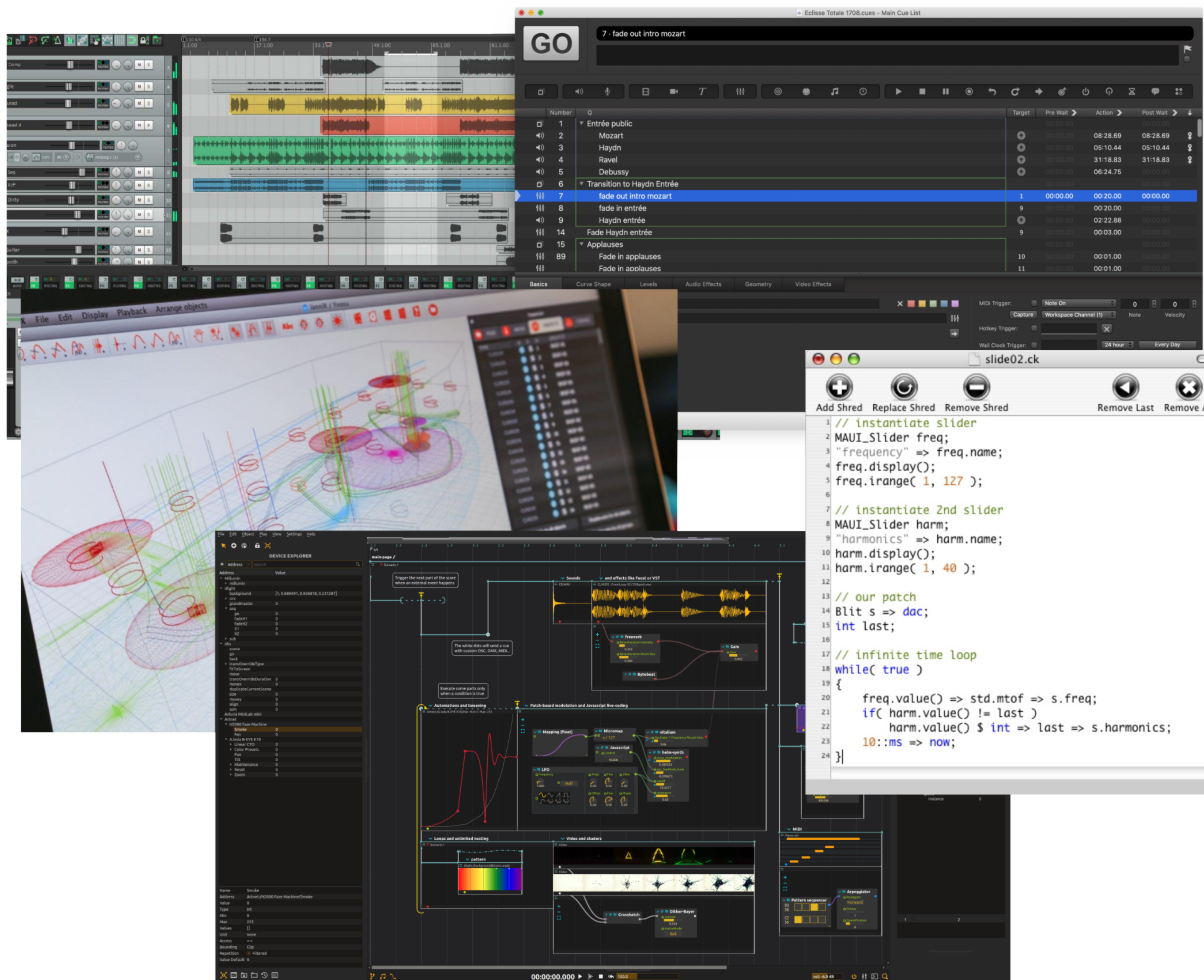



Figure 1.1: Time Metaphors. From top to bottom and left to right: Reaper, Qlab, Iannix, Ossia Score, and ChucK.

Most show controllers also feature timelines, but allow multiple timelines to be triggered independently and run in parallel, and give more dynamic control over the placement of cues along timelines. Examples of such software include QLab (Figure 53, n.d.), Medialon (Medialon Ltd, 2019), or Smode (Smode Tech, n.d.).

### Timed Graphs

Ossia Score (Celerier et al., 2015) organizes cues on a 2D canvas where time can flow from left to right along multiple timelines. However it also features logic and reactive elements, and its underlying temporal model is inspired by Petri nets formalism, which makes it closer to our timed graphs category.

Patching environments built around reactive dataflow graphs such as Max (Cycling 74, 1997) or PureData (Puckette, 1996) are sometimes used to encode complex temporal scenarios. Contrary to Ossia, they don't explicitly represent time as part of the canvas display, but instead temporality is encapsulated in delays and "sequencer" nodes (e.g. the `qlist` object).

### Spatial Sequencers

Iannix (Coduys & Ferry, 2004) and Geosonnix (Graham, 2015) are spatial sequencers that use an explicit 3D space metaphor where playhead objects move along trajectories and trigger actions by colliding with other objects. These can be viewed as extensions of the concept of timelines in more than one dimension and with variable playback speed, allowing to somewhat detach the real-time of the performance from the symbolic time of the scenario.

Graphics scores in PureData (Puckette, 1996) are an example of spatial metaphor, where data structures are represented and manipulated as 2D shapes in a canvas.

### Programming Languages

Antescofo (Cont, 2008) is a score following system that couples a listening machine with a synchronous programming language (J. Echeveste et al., 2013). It is based around an augmented score that interleaves expected musical events such as notes, trills or chords, with temporal reactions that execute concurrently in the time of the performance, and can span across multiple logical instants. This model has proven useful for composing and performing mixed music works, since it allows describing musical processes that evolve over time and interact with the human player, rather than mere

instantaneous reactions. Antescofo also allows composing independent abstract times through the use of *time scopes* and tempo curves. Antescofo is embedded in a runtime context such as Max or PureData, that handles the multimedia synthesis aspect of the performance.

ChucK (Wang et al., 2015) is an audio synthesis environment where the configuration and parameters of a node-graph synthesizer are driven by a domain specific synchronous language.

Gibber (Roberts et al., 2014) relies on a general purpose scripting language (Javascript) and notation conventions to build live-coded multimedia performances.

### 1.1.2 Abstract Time and Poly-Temporality

Despite the variety of approaches, most tools lack an abstract notion of time, as they directly map cues to clock-time dates or to external triggers (a notable exception being Antescofo). Some of them provide ways to tweak their playback speed and program discrete or continuous tempo changes. However, this is still a global mapping from one single abstract time to clock-time.

Musical time is often deployed throughout a work at different scales (e.g. movements, phrases, cells, notes...). Furthermore, not every scale is tied to the same global tempo, e.g. ornaments such as grace notes and *appogiatura* are not affected in the same way by a change of tempo as a main melody line. Hence it would be more appropriate to allow the use of multiple abstract *timeframes*. This aspect has been tackled early by FORMULA (Anderson & Kuivila, 1990), a computer music performance system featuring a Forth-based synchronous programming language. FORMULA's programs interleave actions and waiting primitives. The system allows applying independent time deformations on groups of concurrent tasks. Time deformations are specified as sequences of coroutines that maintain a time position. FORMULA also features automatic action buffering, allowing to perform computations ahead of time, and thus absorb erratic computation delays and maintain strict timing guarantees. Jaffe (1985) also proposed a recursive scheduler for hierarchical timing control, using explicit time maps.

Ossia allows breaking the timeline into multiple branches and controlling each branch's playback speed with tempo curves. Antescofo has the notion of *temporal scopes* that associates blocks of actions to hierarchical timelines, similar to FORMULA (Giavitto et al., 2017). Time transformations are

explicitly specified as tempo curves or implicitly controlled by the listening machine using various synchronization strategies.

## 1.2 Programming Interfaces

Somewhat distinct from the temporal models exposed above, we can observe a spectrum of programming concepts representations and interfaces, ranging from *symbolic* to *figurative* approaches, in the sense specified below:

- In *symbolic* approaches, semantics are encoded in some kind of structure built on discrete static symbols. Entities in the program are constructed by combining symbols. They are also typically referenced using symbols, which tends to dissociate locality of representation and locality of effects, e.g. proximity of code segments can be irrelevant to their temporal relationships or data dependencies. The representation of symbols themselves don't convey much information, what matters is their association to the symbols they can combine with and the entities they refer to.

- In *figurative* approaches, semantics are manifested and manipulated through visual (and often conceptually continuous) entities such as curves, progress bars, playheads, etc. Representations of entities that maintain temporal or data dependencies are also generally related by explicit visual elements or spatial layout.

We prefer this spectrum to the more obvious textual/visual dichotomy, which we don't find very operative. Beyond the ambiguity of the term *visual programming*[1] it doesn't seem to yield much insight about specific properties

[1] As trivial as it may sound, most textual systems are still represented visually, with glyphs displayed on a screen. Conversely, a lot of "visual" systems still convey a lot of information textually. Burnett (1999) proposes a definition of visual programming that seem to avoid these naive objections: *"Visual programming is programming in which more than one dimension is used to convey semantics."*.

However, this doesn't seem to be a more actionable definition to us. For instance, dimensionality is not really used to *convey semantics* in a node graph, nor in a spreadsheet: these representations are just laid out in two dimension by necessity of being displayed on a flat screen, as much as text representations.

Maybe dimensionality here is to be understood in a more conceptual sense, but conceptually both text programs and node graphs really represent graphs. They don't really have a dimension before being projected onto a flat screen. In that context, the unidimensionality of text only pertains to its underlying *encoding*, as a stream of bytes. But that seems to conflate the medium with the message. And after all, the underlying encoding of a node graph is also a unidimensional sequence of bytes!

of the languages in each category. Indeed, among the "visual programming" environments, we can observe a diversity of paradigms, some of which seem really closer to the symbolic nature of textual languages than to other, more figurative approaches. By contrast, symbolic and figurative means of conveying programming concepts can coexist in the same environment, but each approach comes with its own set of practical tradeoffs.

We mention a few paradigms traditionally thought of as "visual" in the following non exhaustive list and try to disentangle the symbolic and figurative elements in each of them. Figure 1.2 also shows relevant screenshots for each category.

- **Patching Environments.** In patching environments, or node graphs, the basic operations are exposed as a set of black boxes *nodes* with inputs and outputs. Programs are specified by building a graph of nodes connected by patch cords on a 2D canvas. The patch cords (i.e. the graph edges) can represent either data flow or control flow between nodes.

  Some examples in this category are: Max (Cycling 74, 1997), Pure-Data (Puckette, 1996) for audio/video synthesis, OpenMusic (Bresson et al., 2011) for computer aided composition, JangaFX's EmberGen (JangaFX, 2019) for fluid simulations, Blender's node graphs (Blender Foundation, 1998) for 3D modeling and shading, or Unreal Engines' Blueprints (Epic Games, 2014) for games scripting.

  Patching environments are often considered "visual", because users interact with the underlying node graph through a graphical representation. However, the spatial layout of most patching systems (i.e. the position of boxes) doesn't carry much information[2]. What matters is the relations between the discrete nodes, whose semantics are mostly specified using symbolic identifiers. Patch cords, which are the semantically meaningful visual elements, only carry information as much as they relate two symbolic nodes. As such, the visual nature of patching environments isn't of much significance. What distinguishes them from textual programming languages is that the control and/or data flows are conveyed figuratively rather than symbolically.

[2]Max makes the peculiar choice of using box positions on the horizontal axis to disambiguate the order of execution when an output is patched to several boxes: boxes execute right to left. Patches can still be spatially rearranged in a large number of ways without that having any effect on their semantics

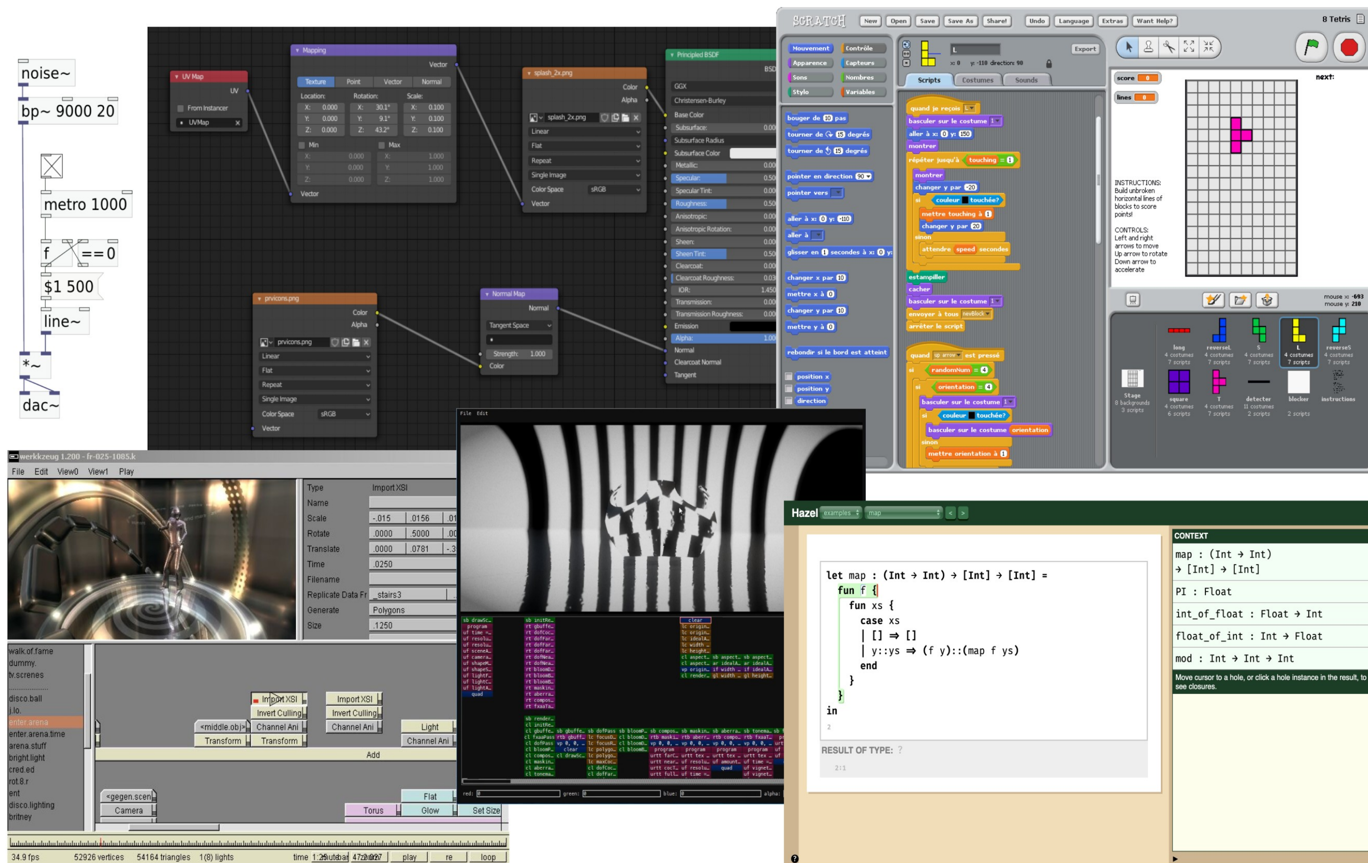



Figure 1.2: Non-textual Programming Interfaces. From left to right and top to bottom: PureData, Blender shader editor, Scratch, .werkkzeug, Demotivation, Hazel.

The analogy with physically patching electronics modules and the explicit visualization of data or control flow between nodes makes these environments intuitive for technical but non-programmer users such as multimedia artists. They also generally offer a lot of domain-specific functionalities out of the box, and can be extended by downloading packages adding new nodes to the environment. Some drawbacks of patching environments are the visual clutter produced by complex graphs, and the unnecessary manual work of laying them out in 2D and connecting boxes together. The irrelevant degree of freedom in the box positions also means the exact same program can look vastly different just by changing its layout.

- **Operator Stacks.** Some demoscene[3] production tools, such as Farbrausch's .werkkzeug (Farbrausch, 2011) or Ferris' Demotivation (Ferris, 2017), address the node graphs drawbacks by stacking operators with matching outputs and inputs on top of each other to create pipelines, and snapping them to a fixed grid. Control and/or data flow from top to bottom, with occasional jumps to other stacks to implement a form of subroutine. This operator "sheet" typically executes each frame to produce a graphic output, and dynamic values are controlled by a timeline or an external tracker[4]. The flat sequential structure of operator stacks and the jump-based control flow of these demo tools feels surprisingly close to writing assembly for a very specialized instruction set. Although they are "visual" in their presentation and by the nature of their output, they seem less figurative than node graphs and closer to the symbolic model of textual programming languages.

- **Structure editors.** Structure editors let users view and edit structured data, using specific knowledge of the underlying format. This is to contrast with text editors, which operate on mostly unstructured

[3]The demoscene is an informal computer art community that rallies around the production and watching of *demos*, i.e. computer programs that generate audiovisual presentations, often trying to push the boundaries of what is feasible under technical constraints. Examples of such constraints include making demos of limited executable size, such as 4K and 64K *intros* (i.e. executables whose size is less than 4 or 64 kibibytes), or targeting old school platform such as the Commodore 64 or Amiga computers.

[4]A tracker is a software that is used to compose and play music using commands organized on a temporal grid. Commands include playing pre-computed samples or waveforms, setting the volume, modifying the playback speed, applying pre-determined effects, etc. Trackers are often used to produce chiptune, a musical genre whose aesthetics revolves around chip-generated sounds and low-resolution samples.

streams of bytes, even when those streams are meant to encode some structure, such as a program. Word processors or cell sheets are good examples of widely used structure editors. Although structure editors mostly gained wide adoption *outside* computer science academia[5], there is a long history of research into structural code editing:

- Early syntax-directed editors, such as MENTOR (Donzeau-Gouge et al., 1980) or the Cornell Program Synthesizer (Teitelbaum & Reps, 1981) present users with menus allowing them to select valid language constructs to insert at each stage of the editing.
- Editor generators explored by the Gandalf Project(Habermann & Notkin, 1986) were meant to automatically generate syntax directed editors from language grammars. A modern descendant of this approach is JetBrain's MPS (JetBrains, n.d.), which allows building custom code editors with language-aware features like syntax highlighting and auto-formatting, and special-purpose graphical user interface widgets.
- Cell sheets are ubiquitous in office work and arguably the most successful (in terms of adoption) end-user programming paradigm. People from a wide variety of professions and educational backgrounds, but who are not primarily programmers, use cell sheets to carry on their day-to-day tasks, sometimes creating fairly complex mini-applications in the process.
- In block-based languages like Scratch (Scratch Foundation, 2003), users can drag and drop blocks featuring jigsaw-puzzle-like tabs and blanks from a block palette, and snap them with other matching blocks in a coding area to form a program. These environments have been successful as teaching playgrounds, although the drag and drop interaction can quickly get tedious for programs of non-trivial size.
- Research rooted in type theory and functional programming has lead to structured editing with "typed holes", as demonstrated in Hazel (Omar et al., 2017), or "tiled editing", as proposed in Tylr (Moon, 2022).

[5] In 2022, most computer science papers and theses, including this one, are still being written in LateX.

  - Some structural editing research has lead to attempts at designing general purpose structured formats such as Dion (Dion Systems, n.d.) or Infra (Hall et al., 2017).

It is important to point out that some aspects of a tool could be considered figurative while other aspects of the same tool could be considered more symbolic. For example, with regards to the notion of time, textual languages and patching environments would fall on the symbolic side, whereas spatial sequencers and show controllers would mostly fall on the figurative side, with Ossia somewhat straddling the line between the two categories.

Purely symbolic approaches lack dedicated input affordances[6] to manipulate continuous processes and time transformations. They also generally fail to convey the abstract temporality of the scenario, and lack live feedback paths to inform users about the concrete temporality of the performance. By contrast, figurative approaches can provide dedicated input methods, allow better visual intuition at a glance, and can embody live feedback, through animated spatial mappings (e.g. the arrangement of elements on a canvas or 3D space, trajectories, sliders, function plots, playback cursors, progress bars, etc.).

On the other hand, figurative approaches are usually locked into a narrow expressivity range. The abstraction floor is set by their basic visual building blocks. This is somewhat constrained to be relatively high to avoid an explosion in visual clutter (e.g. in node graphs) or interaction effort (e.g. in drag and drop block-based languages).

There is also a relatively low expression ceiling because these approaches generally have limited procedural abstraction mechanisms, precisely due to their figurative nature. For instance, different parts of the program can't *independently* call into the same node and get a result, because the inputs and outputs of that node must be wired to the different call sites. In a pull model (where nodes that need to perform some computation pull their inputs from upstream nodes), that would mean the called node would pull

[6]The term *affordance* was coined by James J. Gibson in the context of ecological psychology (Gibson, 1977), to denote "what [the environment] offers to the animal, what it provides or furnishes". It was later borrowed and applied to design by Don Norman in *The Design of Everyday Things* (Norman, 1988). In this context, it is used to designate physical or virtual objects that offer and advertise some possibility of action to a user (one can say that it *affords* that action). To take a classic example, a door plate is an affordance because it suggests and makes possible the action of pushing the door (i.e. it affords pushing). By contrast, a door handle affords both pushing and pulling, which can be more confusing.

data from *all* its upstream nodes. In a push model (where nodes push their output to downstream nodes), the result would be pushed to *all* downstream nodes.

Indeed, for some computation to happen in a given data (or control) path, it has to *be there* in the path, figured by a box. This is to be contrasted with the level of abstraction provided by a procedure, whose code is defined outside any particular code path, and that can be called independently (in term of data or control flow), from multiple places. Instead reusability is usually carried out through creating multiple stateful instances of a template object or patch.

The coarse granularity of primitives also limits the ability to build solutions that tightly fit a problem by composition of building blocks alone. Furthermore, extending the environment is often difficult or impossible short of writing a new building block in a completely different language ecosystem.

## 1.3 Conclusion

In this chapter we reviewed several multimedia production tools and programming environments, both in term of their temporal representations, and their programming interfaces. We attempted to define a rough classification of temporal metaphors, and showed how they are used and sometimes combined in existing tools. We questioned the opposition between textual and visual programming, and proposed to discuss programming environments in terms of symbolic versus figurative interfaces, which in our opinion better accounts for the diversity of approaches.

The survey of existing tools for the authoring and performance of live show scenarios reveals some sparsity in the solution space. First, there aren't many tools that propose a robust notion of abstract time and poly-temporal scenarios. Second, there seem to be a lacking bridge between traditional programming languages, with their lack of input affordances and feedback paths on one hand, and spatial or node graph approaches, with their visual clutter and abstraction issues on the other hand. Cuelist software, as a third alternative, have the potential to provide a flexible non-linear temporal model, but have very limited programmability. These observations hint at the potential of a hybrid solution coupling the benefits of the symbolic approach of a programming language with specialized figurative interfaces supported by a structural editor.

Before venturing into that path, this review prompted us to explore the solution space with two small prototypes, probing both ends of the symbolic/figurative spectrum, which we discuss in the next chapter. This preliminary work in turn helped us refined the distinctions between the aforementioned approaches, and seeded some of the ideas which later developed into the Quadrant environment.

# Chapter 2

# Preliminary Work

In this chapter we discuss some preliminary work we did on two proof of concepts, each sitting on one end of the symbolic/figurative spectrum:

- An interpreted textual language with first-class concurrency and temporal semantics, presented in section 2.1.
- A visual show-controller with hierarchical cuelists and tempo curves, which we discuss in section 2.2.

These exploratory steps helped us shape the symbolic/figurative distinction, as well as clarify its tradeoffs. Ultimately, it determined us to explore ways of combining the strengths of these two categories in a new environment. It also helped define the design goals of such an environment and test ideas for its temporal language and user interface.

## 2.1 QScript: A Textual Temporal Scripting Language

Our first explorative step was to build a textual language, dubbed QScript, seeking to express open temporal scenarios using synchronous, concurrent streams of execution and hierarchical symbolic time frames. Although the Antescofo language (J. Echeveste et al., 2013) provides the constructs to express such scenarios, we felt the need to depart from it in a few important ways:

- We wanted the language and its interpreter to be detached from the concerns of score following, which is at the root of Antescofo's design choices.
- Much of the peculiarities of Antescofo's syntax comes from the constraint of being compatible with Max messages syntax. We wanted to lift that constraint and explore simpler, more consistent syntaxes.
- We wanted to favor static type checking over the dynamism of Antescofo.

### 2.1.1 Overview

QScript is built around the notion of cue group, which associates a lexical scope and a symbolic time frame. Groups execute concurrently, and map their symbolic time frame onto realtime using a simple scaling factor. Nested groups inherit the combined scaling of their ancestors.

An example of of the language's syntax is given in Listing 2.1.

```
def Note(msg: []char, note: i32, vol: f32)
{
        PrintLine("Send OSC: ", outlet, msg, note + transpose, vol);
        OscSend(outlet, msg, note + transpose, vol);
}

group
{
        $> 0: "bar 0": group
        {
                $init let msg : []char = "/lead/note";
                $> 0.25*beat: Note(msg, 59, 0.5);
                $> 0.5*beat: Note(msg, 64, 0.5);
                $> 0.75*beat: Note(msg, 66, 0.5);
        }
        //...
}
```

Listing 2.1: QScript statements example

The language statements (also called cues) are composed of three parts tiers:

- An optional time expression consisting of a temporal connector and a delay expression.

- An optional label naming the cue.
- An instruction. Instructions can be simple instructions like affectations or procedure calls, control flow instructions such as conditionals and loops, or temporal groups that launch a new concurrent stream of statements.

Informally, the delay in a time expression specifies a duration to wait before the execution of the statement, and the temporal connector defines an anchor point from which to start waiting. The anchor point is relative to the execution of other cues, so in order to define the semantics of cues, we first need to describe the a number of properties related to the temporality of statements execution. We then define specific time points of interest in the execution of a cue, and finally explain how time operators match the anchor point of their cues with the execution of other cues.

### 2.1.2 Temporality of Statements

The execution of statements is normally "inserted" in the timeflow of their parent group and participate to its execution time. However, if a statement S is itself a group, the path of execution is forked and S is executed concurrently with the other cues of the parent task. We say that the path that executes S is the forked path, and that the one that continues executing the following cues is the main path. We say that an executable "element" (which can be a statement, a procedure, or an expression) is immediate if no potential main path of execution of that element contains a time expression. In other words, no potential path of execution may contain a time expression, unless after the execution of a group statement. We say that it is temporal otherwise (although it may not always wait in the main path). The important distinction here is that all instructions of the main path of an immediate element are guaranteed to execute at the same logical instant.

Each cue has an anchor point, which is the time at which its delay expression is evaluated and the starting point of the wait. Each time operator aligns the anchor point of its cue with some point of the previous cue, or of the parent group. Table 2.1 shows the effect of each time operator on the anchor point.

| | | |
|---|---|---|
| `$>` | absolute operator | The delay starts from the beginning of the group. |
| `\|>` | parallel operator | The delay starts from the same point as the delay of the previous cue. |
| `:>` | follow operator | The delay starts from the end of the execution of the previous cue's statement. |
| `=>` | shallow join operator | The delay starts from the retirement of the group launched by the previous cue's statement. |
| `+>` | deep join operator | The delay starts from the completion of the group launched by the previous cue's statement. |
| `$init>` | init operator | The cue is executed at the time the group is launched, before other cues. Delay is ignored. |
| `::` | standby operator | The delay starts from an event triggered from the editor. It is not defined with respect to the previous cue's execution. |

Table 2.1: Time operators.

### 2.1.3 Interfacing with Foreign Code and External Software

QScript also has a "foreign import" system that allows loading dynamic libraries and importing symbols, such that QScript programs could call foreign functions as if they were QScript functions. This allows us to use OSC and networking libraries, and to quickly expose their functionalities to QScript. An example of using the foreign import directives is given in Listing 2.2.

We built a demo where we used a QScript program to drive a polyphonic Max synthesizer via OSC messages and play the intro of a chiptune song, using concurrent groups for each voice[1].

### 2.1.4 Limitations

While this early experiment informed some later design decisions, we ultimately chose to not push it further. It had become clear that it was bound

[1] https://youtu.be/j81_Z-i9Yb8

```
// loading a library and importing functions from it:
foreign "bin/corelib.dylib"
{
        def OscOutletOpen(addr: []char, port: u16) -> u64
        def OscOutletClose(outlet: u64) -> int
        def OscSend(outlet: u64, pattern: []char, args: ..any) -> int
}

// foreign functions can be called as normal ones:
OscSend(out, "/voices/lead", pitch, gain);
```

Listing 2.2: QScript foreign system example.

to reproduce the same important issues as other textual languages in that space, such as Antescofo (J. Echeveste et al., 2013), Chuck (Wang et al., 2015), or Super Collider (McCartney, 1996).

As mentioned in Introduction, these languages lack both input affordances and feedback paths tailored to their domain.

- These languages typically deal with conceptually continuous curves (e.g. in the form of tempo curves, trajectories in some parameter space, signals...). Text is obviously an awful affordance for specifying these curves. Operators have to resort to external tools to draw these curves and import textual parameters back into the language, either manually or with often brittle text-replacement. Some of them opt to not use the language curve specification constructs and instead send serialized curve data from an external tool at runtime.
- These languages can't offer any kind of feedback or live debugging capabilities by themselves, again forcing users into devising ad-hoc visualization solutions, often involving a lot of superfluous code instrumentation, message passing, and user interface work.

Domain specific programming environments have the opportunity to tailor their user interface to their domain, which can not only improve the productivity of code literate users, but also has the potential to empower non-programmers to harness those tools in their creative endeavours. In the specific use case of a show-control environment, operators should be able to specify and manipulate tempo or other commonly used parameters through specialized graphical user interfaces, like curve editors, sliders, color pickers,

etc. The environment should also give them useful live feedback about the running scenario, or the time remaining before the execution of a cue, or which code paths are taken by the program. In this context it seemed like a net loss to develop yet another textual programming language and miss the opportunity to tackle these important problems.

## 2.2 QEd: A Cuelist Editor

In a second exploratory step, we developed a functional mockup of a visual show-controller. This tools was built around hierarchical cuelists organized by nested temporality.

### 2.2.1 Overview

The graphical interface of the cuelist editor can be seen in Figure 2.1. Each group (cuelists surrounded by red rectangles) is associated with a timeframe. Each cue represents an autonomous coroutine that executes in the time frame of its parent group. Each type of cue was to be defined as a plug-in in the form of a dynamically linked library calling into the show controller's API. We envisioned several types of cues:

- Audio/video cues for playing back media files at varying speeds.
- Light cues that could send DMX commands to dimmers.
- MIDI or OSC cues allowing message-based communication with other audiovisual softwares.
- Script cues opening the possibilities of end-user programming.

Cues are positioned on their time scale using a delay (indicated by the number displayed in front of the cue name), and operators that specify an anchor point from which the delay is calculated:

- −> started the countdown from the launching of the preceding cue.
- => started the countdown at the end of the preceding cue.
- | indicated that cue was launched manually and didn't depend on preceding cues.

The bottom panel was destined to display the graphical interface of the selected cue (e.g. for an audio cue, the path of the audio file, starting and end timecodes, looping options, etc.).

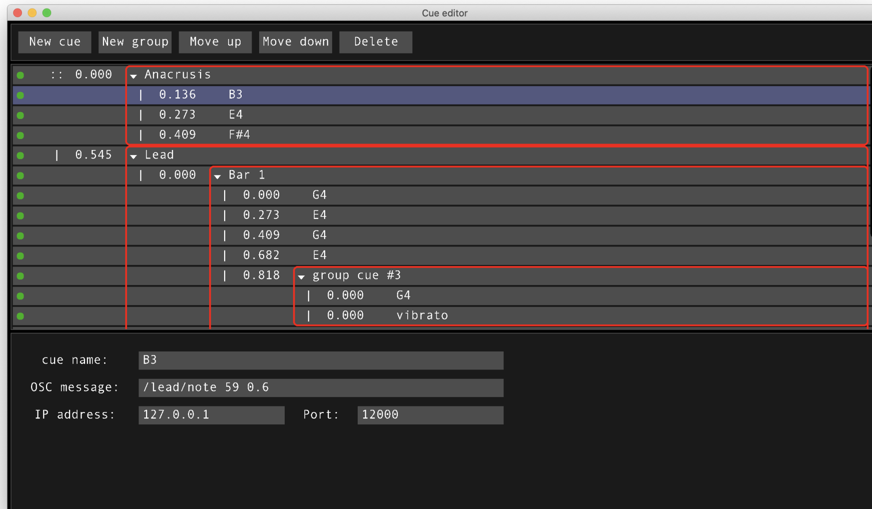



Figure 2.1: Graphical interface of our prototype show controller.

We implemented basic OSC cues and setup a simple scenario similar to the one used in section 2.1, where a cuelist drove a polyphonic Max synthesizer with several parallel groups[2].

### 2.2.2 Limitations

In this model of cuelist-based show-controller, the focus is put on the temporal layout of cues, and most actual computations and side-effects are produced by predefined closed-box cues. This is actually a great strength in a vast majority of cases, where being able to quickly iterate on an easily manageable cuelist hierarchy is more important than a high degree of programmability. However, it severely limits the compositional capabilities of the environment, i.e. the ability to build larger processes by composing smaller ones. Hence these tools can come short in situations where the set of cues alone doesn't quite cover the needs of the show.

The clear delineation between temporal organization and actual cues also means programmatic tools offered to the user are mostly divorced from the temporal features exposed by the environment. Save for some basic jumping

[2]https://youtu.be/R1JGkMWOwBY

features (e.g. cues triggering other cues or cuelist), scenarios are intended to be primarily defined in a visual and declarative way (by organizing cues in cuelists or placing them on timelines) rather than a programmatic imperative one. This can be a problem for highly interactive and/or non-linear scenarios.

In our show controller, cues interfaces and behaviors where to be written as dynamic libraries, readily opening an avenue to extend the environment with new primitives (i.e. new cues). However, this requires the extender-user to deal with a completely separate toolchain and abstraction level. This abstraction frontier, which is faced by other visual programming environments such as Max or PureData, precludes most users from extending the environment. Such environments typically work around this problem by providing one or more script-embedding blocks (e.g. a scripting "node" in node-graphs environments). These scripting blocks typically embed a completely separate interpreter or VM for a textual scripting language like Javascript, Lua, or Python. They also typically feature an in-line text editor and support hot-reloading. We intended to support end-user programming in the same way, through the use of script cues giving access to a dedicated temporal scripting language inspired by QScript (section 2.1).

Although this approach is better than requiring to use a separate toolchain, it still breaks the abstraction continuum by imposing a strong frontier between the concepts exposed by the scripting language and those being manipulated in the graphical environment. Furthermore, each piece of script is encapsulated inside its own opaque block and can only communicate with other scripts through the channels offered by the host environment, which are typically at a higher level of abstraction.

Another problem arises from the temporality of those script cues. In dataflow oriented or reactive environment, the script is run in response to a (possibly recurring) event. In a show-controller, we expect the ability to specify long-running tasks which can be paused and resumed according to a predefined timeline or to external events. Mixing temporality as specified by a long-running script with temporality as displayed and manipulated inside a cuelist manager, in a non confusing way, is an interesting user interface problem. When do the objects defined in a long running script "exist" and can be accessed from the outside? Assuming scripts can share state or communicate in some way, how do we ensure lifetimes stay correct after moving cues and reorganizing the cuelist? How is the internal progression of a script reported in the cuelist user interface?

While these concerns can certainly be addressed, imposing such a strong conceptual barrier between scripts and the rest of the environment inevitably leads to dissociated ways of handling them on each side of the barrier. This creates a steep effort gradient around the point where users have to switch to scripting: climbing down the "abstraction ladder" becomes a costly decision that forces users to change their mental model of the tool. In our experience this results in users trying to force their way with inadequately high-level features, and only using scripting as a last resort, or ignoring scripting altogether. We would prefer to have a much smoother effort-reward curve, allowing users to gradually transform and extend the constructs they expressed in one level to another, should the need arise. In particular, this mean we shouldn't require them to abruptly switch between completely different mental representations of the objects they are manipulating.

## 2.3 Conclusion

The textual language prototype we presented at the beginning of the chapter proved its limitations as soon as we started integrating tempo curves to express time transformation. It also had the obvious drawback of not providing any avenue for live feedback in itself. Building a specialized editor to support these features implies reimplementing a large part of the compiler inside the editor, which leads to question the relevance of keeping them separate. At that point, making a round-trip between a textual source code and a structured representation of the program adds processing stages while providing little benefits.

Coming from the other side of the symbolic/figurative spectrum, our attempts at a small cuelist-based show controller highlighted the limited end-user programmability of such an approach. Straying from the "black box" model of cues and exposing finer granularity constructs would yield higher composability and soften the steps between higher and lower levels of abstraction. In particular it would open the way to extend the environment "from within", without having to resort to a separate toolchain and programming paradigm to create new cues. However, this would likely require introducing ways to reuse and combine user-defined constructs through symbolical means, veering away from the purely figurative nature of the original model. It would also imply a change in the interaction model, since creating complex scenarios from finer constructs via menus or shortcuts would become impractical.

The insights gathered during the development of these two prototypes seem to converge on a common, hybrid approach. Blurring the lines between a programming language and a dedicated editor, would allow introducing custom input user interfaces and live monitoring into a symbolic model. Conversely, breaking up the basic pieces of a show controller into finer-grained building blocks and providing better support for composition and abstraction would bring it closer to the flexibility of a programming language.

The remaining of this work presents our attempt at defining and concretizing such an hybrid approach through the development of Quadrant, a programming environment tightly integrating a structurally edited temporal language along with its compiler pipeline and runtime.

# Chapter 3

# Introducing Quadrant

In this chapter we introduce Quadrant, an integrated programming and runtime environment for temporal scenarios[1]. Quadrant is written in C, on top of a platform layer and user interface toolkit, called MilePost, also written mostly in C. The source code is available in the following `git` repository: https://forge-2.ircam.fr/fouilleul/thesis_quadrant. It has been developed on an macOS 10.15 platform, and although no effort has been expended to port it to other platforms yet, we tried to maintain a clear separation between platform-specific and platform-agnostic code.

In the introduction, we emphasized the importance of abstract time and poly-temporality in live shows scenarios. The preliminary work presented in chapter 2 informed our latter work on a temporal scheduling model supporting these notions. In addition to a review of existing tools and their temporal metaphors, it also helped us shape our understanding of the distinction between symbolic and figurative approaches as it pertains to the authoring and representation of temporal scenarios, as discussed in chapter 1. Assessing the tradeoffs and limitations of both representations lead us to define goals for a new approach that resulted in the Quadrant prototype:

- The new environment should give the fine granularity of control, composition capabilities and abstraction mechanisms of a programming language.

[1] A quick overview of Quadrant is also available in video form here: https://www.youtube.com/watch?v=6nC2M3NwDe8. This presentation was recorded for the Sound and Music Computing Conference 2022, and includes a small demo of a scenario used to generate note sequences and pilot a polyphonic synthesizer

- It should allow expressing multiple abstract timeframes and time transformations.
- Temporal constructs should be specified in the same language context as other programming constructs, allowing to seamlessly flow between timing and computation.
- It should include specialized input systems when necessary, e.g. to describe continuous curves.
- It should provide visual and live feedback information directly in the context where the scenario is specified.
- A secondary objective was to mitigate the cognitive cost of a language based approach, especially for non-programmers, by providing an integrated environment with helper functionalities such as auto-layout, online error messages and completion suggestions.

We first give an overview of the Quadrant environment in section 3.1. We then present the Quadrant structure editor in section 3.2. We discuss static feedback, and execution monitoring in section 3.3.

For the sake of clarity, we focus here on the user-facing part of Quadrant's design, and the improvements it could bring to temporal scenario authoring and monitoring. We only briefly mention its underlying temporal model, its custom language, its compiler pipeline and its runtime. These components are presented in more detail in subsequent chapters.

## 3.1 Overview

Given the desired level of programmability, we thought a syntactic program representation was best suited. However, in order to fullfill the requirements of dedicated input and feedback paths, the user-facing scenario editor, the underlying representation and the runtime engine have to closely cooperate around a common understanding of the programming model. Thus it is preferable to abandon the traditional text-based representation of programming languages, and work at a more structured level.

Our approach revolves around a structured binary format used to encode temporal scenarios in a domain specific programming language. The environment combines a structure editor used to author and monitor scenarios, a compiler pipeline targeting a custom bytecode, and a runtime consisting of a virtual machine and a temporal scheduler. This general architecture is

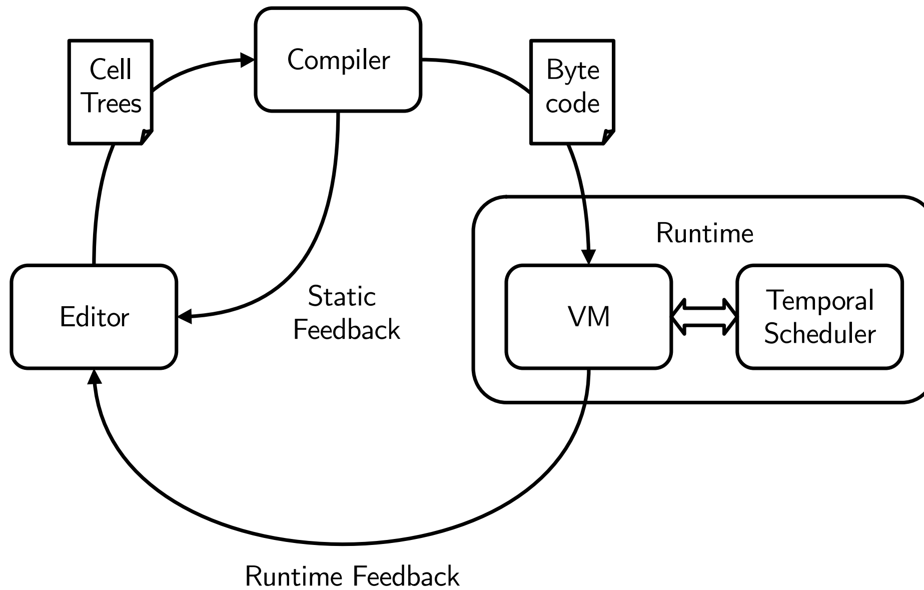



Figure 3.1: Quadrant Architecture

shown in Figure 3.1. The editor operates on a semi-structured representation of the program, which gets fed to the compiler. The compiler transforms it into a bytecode program image and passes it to the runtime where it is executed. There are two feedback paths in this architecture. One operates at edit time and provides syntactic and semantic feedback in order to support language-aware editor features. The other operates at run time, and provides execution feedback in order to support live monitoring within the editor.

### 3.1.1 Interface and Example Program

Figure 3.2 shows an example scenario loaded in the Quadrant editor. A Quadrant scenario can be composed of several modules. The top view of the editor shows the currently edited module. The middle view is a console into which a Quadrant program can log messages. The bottom bar features two pop-up menus that allows selecting the currently edited module, as well as the main module of the program (which recursively pulls in all other modules).

Quadrant

```
(import sync)
(import fmt)
(def start ()
   (var f (future void)
      (flow @(tempo #tempo_curve
                     Tempo Curve Editor
                        1  2  3  4  5  6  7  8  9
                                                       )
         (for (var i i32 0)
              (< i 10)
              (set i (+ i 1))
           (fmt:print "[%i], time: %f" i (sync:get_clock))
           (pause 1))))
   (wait f))
```

```
[0], time: 0.000143
[1], time: 0.920307
[2], time: 1.624986
[3], time: 2.182202
[4], time: 2.660017
[5], time: 3.105289
[6], time: 3.557942
[7], time: 4.054017
[8], time: 4.635970
[9], time: 5.364537
```

```
Edited module: ...p/qed/saves/test_tick.ql     Main module: ...p/qed/saves/test_tick.ql
```

Figure 3.2: Example Program

The example program starts by importing the `sync` and `fmt` modules, which provide procedures for respectively querying timing informations and printing formatted data to the console. The program then defines a single procedure named `start`, which is the entry point of the scenario. This procedure immediately starts a concurrent child task with an associated time transformation, controlled by a tempo curve editor. The task performs a loop which prints a message containing the iteration number and the real time, and sleeps for 1 unit of symbolic time between each iteration. The outer procedure just waits for the task to finish, and then exits, terminating the scenario.

The console displays the output of this scenario. As can be seen with the timestamp of each iteration, the process accelerates, then decelerates, following the specified tempo curve.

We go into more detail about the temporal model and the use of tempo curves in chapter 4, and about the language used in Quadrant in chapter 5.

## 3.2 Structure Editor

The goal of the Quadrant editor is to provide more ergonomic input and figurative representations for 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, text 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.

Quadrant's editor 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 of a few different kinds:

- Numbers (integers of decimals).
- Textual identifiers.
- Operators such as `+`, `-`, etc.
- User interface widgets, e.g. curve editors.
- Lists of other cells:
  - S-expressions, e.g. `(+ a b)`.
  - Attribute cells, e.g. `@(foo)`.
  - Array literals, e.g. `[x y z]`.

### 3.2.1 User Interface Cells

The cell tree can contain visual user interface cells. Instead of having a special shortcut or menu option for each widget, these cells can be created by typing the `#` character, and providing a textual widget identifier. The

editor recognizes the identifier and attaches the corresponding user interface widget to the cell. This allows user interface cells to be added in the same way as list or textual cells, without breaking the input model into several methods depending on the kind of cells. This also makes provision for easily adding future widgets by tying them to new names.

In the current state of the environment the only widget is a basic tempo curve editor (Figure 3.3), which allows specifying the tempo of a block of code with a piecewise function make of cubic Bézier curve pieces. This offers both an easier input model and a better visualization of continuous curves than a purely textual construct. The widget view is collapsed when the cursor is not on the widget's cell. This allows the specification of the curve to stay close to its usage site, while not consuming screen real estate when it is not in focus.

When the mouse is hovering the tempo curve widget, each piece's endpoints are displayed as orange dots. Clicking one of them selects it (turning it to red) and reveals neighboring intermediate control points and their corresponding tangents, in blue. Control points can be dragged around with the mouse, and endpoints can be dissociated to create discontinuous curves. Clicking on a curve splits it in two pieces with a common endpoint at the position of the mouse pointer. Clicking an endpoint while holding the `Alt` modifier removes it and joins the two curves on each side into one, preserving their outer control points. While hovering the plotting area, a blue crosshair is displayed and coordinates are reported at the bottom of the widget to help precise positioning.

In the future, more widgets could be added, such as sliders for quickly adjusting numbers, color pickers for sending RGB values to LED projectors, piano rolls for generating MIDI notes sequences, etc.

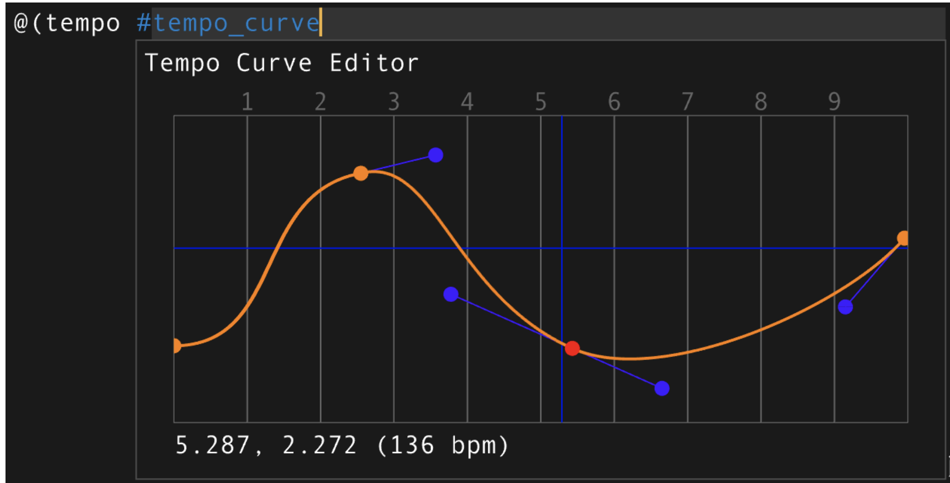



Figure 3.3: Quadrant Tempo Curve Editor

### 3.2.2 Navigation and Selection

The editor maintains a cursor which corresponds to a position inside the tree. The cursor can be described by a triple $(p, l, o)$, where $p$ is the parent cell under which the cursor lies, $l$ is the children of $p$ whose the cursor is immediately left from (and $l$ is 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.

A unique *visual* position of the caret can correspond to multiple positions of the cursor *in the tree*. To help distinguish these different cursor positions, the editor displays the following visual hints: 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.

Figure 3.4 illustrates cursor navigation. The cursor is moved one step left between each image of the figure, adopting the following successive positions:

- The cursor starts between the `s` and the `e` characters of the `set` token, i.e. at position $(\texttt{set}, 0, 1)$. The `set` token is drawn on a lighter gray background to indicate it is the cursor's current parent cell.

- Then the cursor is moved one step left, at the beginning of the `set` token, which corresponds to position $(\texttt{set}, 0, 0)$.
- Next the cursor is moved one step left again, which positions it directly on the left of the `set` token, at position $(\texttt{set-form}, \texttt{set}, 0)$. The entire `set` s-expression is drawn on a lighter gray background, and the cell directly right to the cursor (i.e. the `set` token) is underlined with a green line. In this case there's no child of the cursor's parent on the left of the cursor, so no blue underline is drawn.
- The cursor is then moved left by one step again, putting it directly on the left of the `set` form, at position $(\text{def-form}, \texttt{set-form}, 0)$. The cell directly on the left of the cursor is underlined in blue, and the cell directly on the right of the cursor is underlined in green.
- Finally, the cursor is move left by one step, descending into the procedure signature cell, directly on the right of the `u64` token, which corresponds to position $(\texttt{proc-signature}, 0, 0)$. The cell on the left of the cursor (i.e. the `u64` token) is underline in blue. There is no cell on the right of the cursor to be underlined in green.

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 of consecutive children of $\mathscr{P}$, which contains the point and the cursor (in other words, the first tree of the selection contains the parent of the cursor (or mark), and the last subtree contains the parent of the mark (or cursor)).

This allows growing and shrinking the selection to the next meaningful syntactic boundary as the cursor moves towards or away from the point.

Figure 3.5 illustrates cursor selection[2]. The cursor follows the same path as in the above navigation example, but the mark is left in place by holding the shift key.

- After the first move, the cursor and mark still belong to the same textual token. Hence this result in a text selection covering the `s` character of the `set` token. This selection could be textually copied, cut, or replaced by some other typed or pasted text.

[2] A video of this example is available here: https://youtu.be/rw3KNPb2Ix0

- Next, the cursor and mark are no longer in the same cell. The selection becomes a range of sibling cells, comprising only the `set` token.
- Next, the selection moves one level up to contain the whole `set` form.
- Finally, the selection moves extends left to contain both the procedure signature and the `set` form.

**Implementation details.** The `qed_command` structure, defined in `editor.cpp`[3], bundles cursor moves and actions (such as inserting or deleting nodes) corresponding to a given keyboard shortcut. The `QED_COMMANDS` buffer defines all the commands of the editor. When a shortcut is received, the buffer is traversed, and if a match is found, the `qed_run_command()` procedure is called, which triggers the moves and actions of that command. Cursor moves are implemented by the `qed_move_one()` and `qed_move_vertical()` procedures. Selection and navigation are both implemented from moves, using a `setMark` flag in the `qed_command` structure, which specifies wether the mark must be set to the cursor after the move.

### 3.2.3 Edition

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 `(` character creates a simple list cell, while the shortcut `Cmd + (` encloses the cell directly right to the cursor into a new list cell. These actions are encoded in the `action` field of the `qed_command` structure.

Otherwise, entering text replaces the current textual selection by the input characters, much as in a standard text editor. If the cursor is not inside a text cell, one is created at the cursors' position beforehand. Once the input characters have been inserted, the cell's text is tokenized, which may modify the current cell's kind.

[3]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/editor.cpp

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(a) Cursor is before the second character of the `set` token.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(b) Cursor is at the start of the `set` token.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(c) Cursor is on the left of the `set` token.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(d) Cursor is on the left of the `set` s-expression.

```
/* pseudo-random number generator */
(var prngState u64 45797)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(e) Cursor is on the right of the `u64` token.

Figure 3.4: Cell Tree Navigation.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(a) Selection is empty.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(b) Selection contains the `s` character of the `set` token.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(c) Selection contains the `set` token.

```
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(d) Selection contains the `set` s-expression.

```
/* pseudo-random number generator */
(var prngState u64 45797)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))
```

(e) Selection contains the procedure signature and the `set` s-expression.

Figure 3.5: Cell Tree Selection.

## 3.3 Feedback and Monitoring

As part of compiling a temporal scenario, the compiler pipeline parses and type checks each module's cell tree, which produces an abstract syntax tree and an intermediate representation, including symbol tables and type informations. These syntactic and semantic data structures can then be queried by the editor. It allows the editor to display inline errors, perform syntax highlighting and automatic layout, to automatically create placeholder cells for missing constructs, and to suggest auto-completion bases on the semantic context at the position of the cursor.

While these features can be found in many IDEs and code editors, they generally require deploying and interfacing with language servers, installing editor plugins, or relying on ad-hoc support in the editor. This partially duplicates the effort spent on the toolchain, and increase the overall complexity of such environments. Here, most of the smart editing feature come at a very low cost, since the compiler and the editor share the same “understanding” of language concepts and the same representations of the program.

Metadata produced by the compiler pipeline is also used to map bytecode offsets to cells and enable execution monitoring in the editor.

### 3.3.1 Error Reporting and Placeholder Cells

Errors generated by the parser or the type checker are used to display a squiggly red underline below the faulty cells. When the cursor is positioned on a faulty cell, a contextual box shows a list of errors associated with that cell. For instance, Figure 3.6 shows the error panel displaying a simple type error.

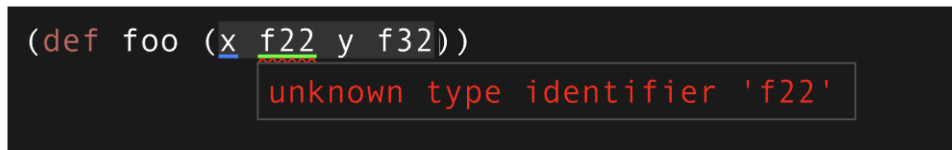



Figure 3.6: Error Panel

Errors for missing cells, emitted from the parser, are used to create placeholder cells to display the expected construct at that location, as shown in Figure 3.7[4]. When the children list of a cell are edited, the trailing place-

[4] A video showcasing placeholders is available here: https://youtu.be/e0JIp8048CM

holders of that cell are removed before parsing. This ensures there are no more placeholder than needed at the end of a cell.

```
(def start ()
   (var variable name type specification))
        missing variable name
```

Figure 3.7: Placeholder Cells

### 3.3.2 Auto-Layout

Before displaying the cell tree, the editor first applies a layout pass. The `qed_update_layout()` procedure in `editor.cpp`[5] is responsible for computing the layout of a subtree. It queries the abstract syntax tree to retrieve the language construct associated with a given cell. Each kind of language construct has a set of associated layout options:

```
typedef struct cell_layout_options
{
        cell_layout_orientation preferredOrientation;
        i32 inlineCount;
        i32 alignedGroupCount;
        i32 alignedGroupSize;
        i32 indentedGroupSize;
        bool endGap;
} cell_layout_options;
```

The layout algorithm proceeds recursively as follows:

- The algorithm recurses into each children and computes its bounding box. Children are laid out horizontally on a single line.
- If `preferredOrientation` is vertical, or the summed width of children exceeds a set threshold, the algorithm proceeds to relayout children vertically, as follows:
  - A first group of `inlineCount` children are laid out horizontally on the first line.

[5] https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/editor.cpp

- The following children are laid out horizontally into a maximum of `alignedGroupCount` groups of `alignedGroupSize` cells. These groups are vertically aligned.
- The remaining children are laid out horizontally into groups of `indentedGroupSize` cells. These groups are indented with respect to the parent cell.

- If `endGap` is true, an empty line is skipped after the parent cell.

A special case is attribute cells, which are always laid out on the same line as the previous cell. Two examples of cell layouts are given in Figure 3.8.

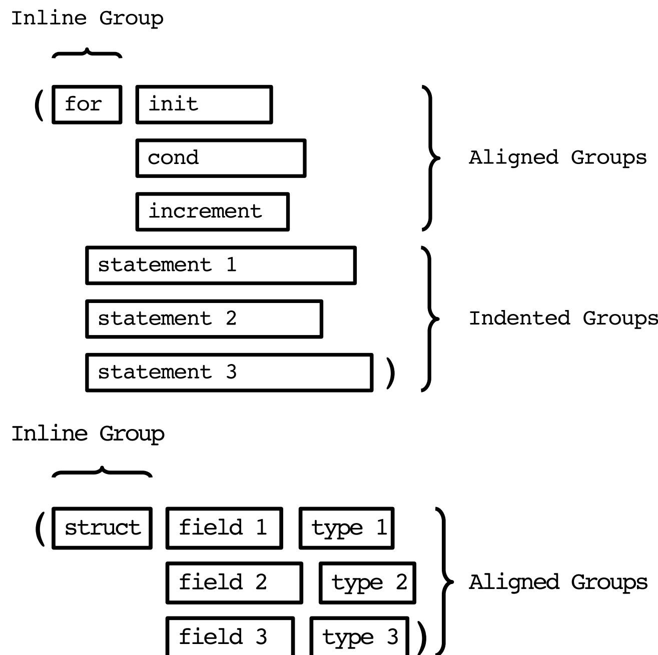



Figure 3.8: Examples of Cell Layouts

The tree is then rendered using a depth-first traversal. The abstract syntax tree data associated with each cell is used to perform syntax highlighting.

### 3.3.3 Code Completion

The abstract syntax tree, as well as the symbols and type representations produced by the compiler pipeline, are both used to drive a simple completion suggestion system.

In each abstract syntax node, the parser stores a "parse rule" identifier, which corresponds to a set of alternative language productions that could be found at that location. Each parse rule is associated with a number of completion patterns, defined in `completion.h`[6]. A completion pattern can directly encode a construct of the language, such as a `for` loop or an `if` statement, or it can match a set of possible completions, for instance "all the variables visible from the local scope". For such sets it also encodes a type pattern, that can match a single type or a set of types, such as "arrays" or "slices".

The completion system is activated by pressing the `tab` key. When active, editing a text cell runs the `qed_completion_update()` procedure defined in `editor.cpp`[7] after the parsing and checking stages. This procedure is in charge of populating and filtering a list of possible completions based on the user input. The completion can apply either to the text cell being edited, or to its parent if the text cell is the head of a form cell. In the following we refer to that cell as the completion cell.

- The procedure chooses the completion cell depending on the kinds of the text cell and its parent.
- The procedure queries the abstract syntax node corresponding to the completion cell.
- The procedure gathers the completion patterns for the parse rule stored in the abstract syntax node.
- Fixed completion patterns that match the kind of the completion cell are added to the completion list.
- If one of the completion patterns is a variable, the procedure gathers all variable symbols visible from the local scope, and matches their type against the type pattern of the completion entry. Matching variables are added to the completion list.

[6]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/completions.h
[7]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/editor.cpp

- If one of the completion patterns is a call, the procedure gathers all procedure symbols visible from the current module and matches their signature against the type pattern of the completion entry. Matching procedures are added to the completion list.

The completion list is then filtered by the text of the edited text cell. Currently, we only keep completions when the input text is a prefix of the completion's text, but one could implement more complex fuzzy matching methods.

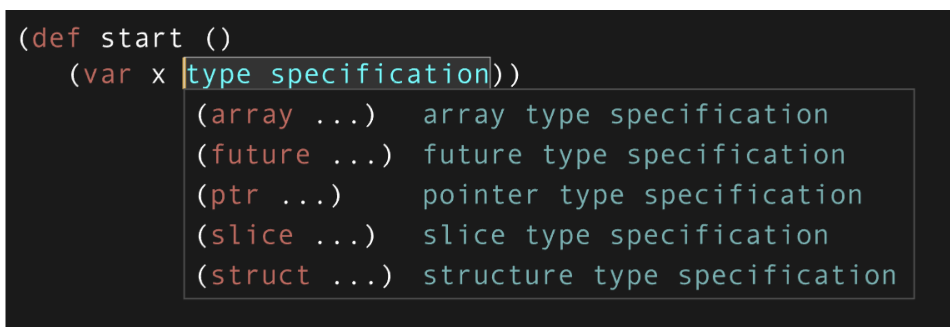



Figure 3.9: Completion Panel

The filtered list is then displayed in a completion panel under the completion cell using the `qed_completion_gui()` procedure in `editor.cpp`[8]. The completion panel is shown in Figure 3.9[9]. The user can navigate through the completion list with up and down arrow keys. The selected completion temporarily replaces the completion cell. Pressing the up arrow key while at the beginning of the list goes back to the original completion cell. Completions can be accepted or dismissed with several shortcuts:

- The `enter` key accepts the selected completion and closes the completion system.
- The `esc` key reverts to the original completion cell and closes the completion system.
- The `tab` key accepts the selected completion, moves the cursor to the next cell, and updates the completion list for that cell.

[8]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/editor.cpp

[9]A video showcasing code completion is also available here: https://youtu.be/kjYfcrSrFKs

### 3.3.4 Live Performance Monitoring

The execution of a Quadrant program can be monitored and paced from within the editor, as shown in Figure 3.10. The following visual hints are displayed by the editor:

- Blocks of code are highlighted with a fading green background as they are being executed, as seen in Figure 3.10a. This allows quickly seeing what code paths are executed and understanding the temporality of execution at a glance.
- When a task is paused, the editor displays a progress wheel in front of the pause instruction (Figure 3.10b). The fraction of the wheel colored in green indicates the ratio of time elapsed in the pause over the total pause time.
- The editor shows waiting icons when a task is waiting for another task to complete (Figure 3.10c). The two green arcs spin as long as the task is suspended.
- When a task is suspended by a `standby` instruction, the editor shows a pulsing standby icon in front of that instruction. When the cursor is positioned on that instruction, a shortcut can be used to resume the suspended task from the editor.

Progress wheels, wait wheels and standby icons are also mirrored in front of every call sites along the call stack of the suspended task. For instance, if a procedure `foo` calls a procedure `bar` that uses a `pause` instruction, a progress wheel will be displayed in front of the `pause` instruction, as well as in front of the `bar` and `foo` calls.

We explain how runtime feedback and the triggering of standby tasks are implemented in section 6.3.

## 3.4 Conclusion

In this chapter, we stated the design goals of Quadrant, as resulting from our prior work on temporal scenarios authoring systems. We then gave an overview of the environment's architecture and its interface, and walked through a simple example program. We then discussed the notion of structural editing, and presented Quadrant's structure editor and non-textual code representation. We finally described the static and runtime feedback

```
(flow
   (for (var i i32 0)
        (< i 10)
        (set i (+ i 1))
      (pause 1)))
```

(a) Flashing code blocks and progress wheel.

```
(var f (future void) (flow
                      (pause 10)))
```

(b) Progress wheel.

```
(flow
   (wait f))
```

(c) Waiting wheel.

```
(standby)
```

(d) Standby icon.

Figure 3.10: Quadrant Execution Monitoring

paths of the environment, which comprises the language-aware editing features of the editor, as well as its live monitoring capabilities.

We only implemented a few of the possibilities afforded by the proposed architecture. Indeed, the tight collaboration of the structural editing, the compiler pipeline, and the runtime around shared program representations opens up a large space of user experience experimentations.

- Code completion could be extended with fuzzy matching. Another improvement would be allowing user-defined completion snippets. For instance, a user could select a code snippet containing holes, and copy it to a user completion repertoire. This snippet would then show up in the completion panel when its outer-most syntactical construct and type would match the cursor location's context.
- We already mentioned a few possible input affordances such as sliders, color-pickers, music staff or piano roll editors, etc. An important and

interesting challenge, that we didn't attack here, would be to allow users to define custom input widgets from within the environment, using normal language constructs.

- These specialized widgets could also convey runtime feedback, such as moving playheads. They could also serve as dynamic viewers for program variables, as opposed to only being initializers, as is currently the case.
- Live program monitoring could also be extended to include debugging and inspection tools, not only as separate watch windows as in classical debuggers, but as live inline tooltips. One could imagine displaying a value inspector tooltip when hovering variables, or highlighting variables with a flashing background when their value is updated. Call stacks could be shown when hovering a procedure's call site.
- Profiling feedback could be displayed as code heat maps indicating performance hot-spots, and performance counters tooltips could be shown when hovering procedure names.
- The environment could record temporal traces of execution and display them as live-updating time plots, to give a visual overview of the scenario's temporality.

In the the following chapters we will go in more detail over temporal and programming model of Quadrant, as well as its compiler pipeline and runtime.

# Chapter 4

# Temporal Model

In this chapter we present the temporal model and the scheduling engine that support Quadrant's temporal features[1]. Our temporal model allows scheduling computations along concurrent timeframes organized in a temporal hierarchy. Timeframes are related to their parent through time transformations. A time transformation can be specified by a tempo curve, either as a function of time or as a function of symbolic position. Piecewise tempo curves can be built from parametric curves such as Béziers curves, which are both versatile and intuitive. The scheduler exposes an interface based on fibers, that makes it easy to organize inter-dependant streams of related events.

We first highlight the importance of symbolic time in musical applications (section 4.1). We then cover the notion of time transformations, and give a differential equation formulation to tempo curves (section 4.2). We then show how we derive and solve tempo curves equations (section 4.3). Finally, we present the symbolic time scheduler used by Quadrant (section 4.5).

[1] This section is based on a paper published in the proceedings of the 15th International Symposium on Computer Music Multidisciplinary Research (Fouilleul et al., 2021). In this paper, Quadrant's scheduler was referred to as "Jiffy", and conceived as a standalone library. Since we moved to a more integrated environment, we dropped the name and simply refer to it as Quadrant's scheduler.

## 4.1 Symbolic Timescales

Despite its importance in music and other performing arts, most show controllers and computer music environments lack an abstract notion of musical time, and directly map cues to wall-clock dates or to external triggers. However, musical time is a symbolic notion that can have many different concrete, real time instantiations. For example, 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 (real) time by the interpreter, following tempo indications, cultural conventions, and interpretative choices. Furthermore, is often deployed throughout a work at different scales (e.g. movements, phrases, cells, notes…), and not every scale is tied to the same global tempo, e.g. ornaments such as grace notes and *appogiatura* are not affected in the same way by a change of tempo as a main melody line. Hence, a temporal programming environment for live shows should allow encoding and performing streams of actions embedded within multiple symbolic musical times, or *timeframes*.

The notion of hierarchical symbolic timeframes has been tackled before by computer music environments or score followers. For instance, FORMULA (Anderson & Kuivila, 1990) allows applying independent time deformations on groups of concurrent tasks. David A. Jaffe (Jaffe, 1985) proposed a recursive scheduler for hierarchical timing control, using explicit time maps. Antescofo (Cont, 2008) allows users to compose independent abstract times through the use of *time scopes* and tempo curves.

In Quadrant, a timeframe is a data structure used to maintain a notion of logical time, expressed as a rational number of *symbolic time units*[2], and to schedule events at specific logical dates. It is analogous in this respect to a score, which organizes musical events in terms of a musical time, that needs to be translated into wall-clock time by a musician according to tempo indications and interpretative choices.

However, whereas the tempo indication of a score usually prescribes some idealized mapping from musical-time to wall-clock time, a timescale's logical time does not necessarily map directly to wall-clock time. Instead, each timescale has a *time source*, which can be either the wall-clock time or an-

[2] We deliberately avoid the term *beats* here. We think it would bring some confusion by conflating the notion of time unit with the notion of meter, and by suggesting that all beats are of equal conceptual length. This is, in fact, rather a Western exception than a universal norm.

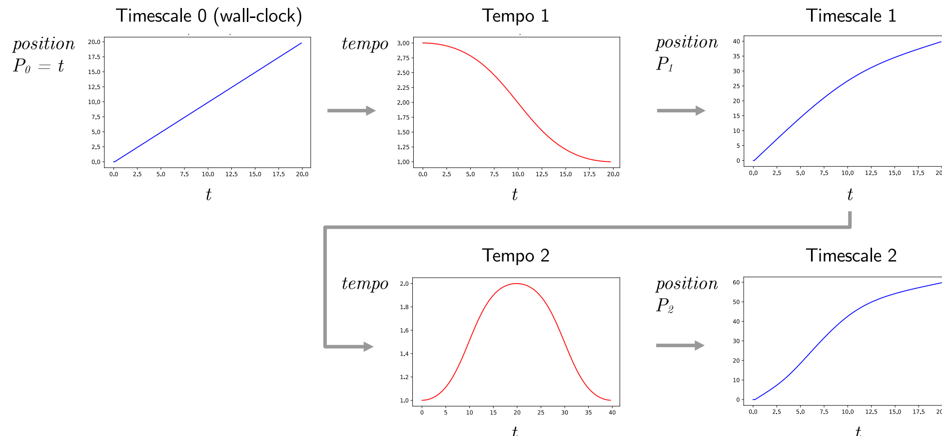



Figure 4.1: Composing time deformations using tempo curves.

other timescale. A timescale is also associated with a *time transformation*, which maps its internal time to the time of its source. Thus, the scheduler can handle multiple notions of logical time and map dates to wall-clock time through a hierarchy of time transformations.

Figure 4.1 illustrates a time deformation between a timescale and its source. The time map plots for each timescale show the position of the timescale with respect to wall-clock time. The effect of the first tempo curve (tempo 1) is to warp the time map of timescale 0 (which represents wall-clock time) into that of timescale 1 (which represent some abstract musical time). Timescale 1 is then transformed by another another tempo curve (tempo 2), to produce the time map of timescale 2.

## 4.2 Time Transformations

Quadrant's scheduler must be able to transform timescale-local positions to and from wall-clock time. These time transformations can be specified as *time maps*, which directly map the parent's timeframe to the local timeframe. The can also be expressed by the means of *tempo curves*, which describe the speed of a timescale's "playhead" with respect to the source time, much like

tempo indications in a score prescribe an idealized conversion from durations in beats to durations in wall-clock time[3].

Tempo curves have been the subject of some controversy. Desain and Honing (Desain & Honing, 1993) claimed that there is *"no abstract tempo curve in the music nor is there a mental tempo curve in the head of a performer or listener"*. They even deemed tempo curves *"harmful"* and a *"dangerous notion"*, basing their opinion on the fact that they are not sufficient to accurately preserve musical timing of a piece under global speed transformations. Mazzola and Zahorka (Mazzola & Zahorka, 1994) firmly opposed this position, arguing that *"The problem is not the a priori concept of tempo curves, but rather its elaboration for realistic application"*.

From a practical standpoint, the debate on the existence of tempo curves seems a little dubious to us. Tempo curves have proven to be of practical use in virtually any music sequencing software. That they do not preserve musical timing when arbitrarily stretched or compressed is not a rebuttal of their adequacy as a compositional tool. In our opinion, a more interesting take on this problem is given by Honing in (Honing, 2001), where he suggests adjoining a time shift map to the tempo curve, in order to independently capture timings that do not react linearly to global tempo changes.

Several methods have been proposed to represent time transformations and to integrate tempo curves to map symbolic position to time. Jaffe (Jaffe, 1985) proposes to directly use time maps constructed from a collection of predefined time warping functions. Berndt (Berndt, 2011) chooses to represent tempo curves by potential functions of symbolic position, matching some specified mean tempo condition. Timewarp (MacCallum & Schmeder, 2010) is a tool that uses regularized beta functions to define tempo curves satisfying polyrhythmic constraints. Antescofo uses a variety of tweening functions[4] to express tempo as a function of time, and uses closed form expressions to compute time transformation based on tempo curves. When there is no analytical solution to a tempo curve integration, Antescofo samples the curve to produce a piecewise linear approximation, which is then integrated analytically. Antescofo can also use arbitrary expressions to define tempo, although these expressions are not integrated: they are reevaluated each time a vari-

[3]One difference, however, is that we use the word tempo here to refer to the ratio of internal symbolic time units over source symbolic time units, rather than the number of beats per minutes, since the latter could depend on the musical meter of the timescale.

[4]https://antescofo-doc.ircam.fr/Reference/compound_curve/

able is updated, and considered constant between updates. As such, they can only represent tempo as step functions.

In Quadrant's scheduler, tempo curve can be specified either as a function of a timescale's source time, or as a function of symbolic position (which is closer to the way tempo is specified in a score). We use piecewise tempo curves where each piece can be defined by parametric curves. Instead of restricting tempo to predefined curves with known integrals, Quadrant uses a variable-step numerical solver to integrate tempo curves when simple analytical solutions are not readily available. This allows us to construct tempo functions from Bézier curves, which are more versatile than standard tweeners and allow easy tweaking of control points by a user through a graphical interface.

### 4.2.1 Differential Equation Formulation

In the following we will use the variable $p$ to denote the position in a timescale, i.e. the logical time in this timescale's reference frame. The variable $t$ will be used to denote the source time (or simply, *time*), i.e. the time in the timescale's parent reference frame (which could be the wall-clock time).

The function *position function*, $P(t)$, transforms the source time into the internal position of the timescale. The *time function*, $T(p)$, transforms the position into the source time. Obviously, $P = T^{-1}$.

A *tempo curve* $\boldsymbol{\mathcal{T}}$ can be either a function of time or position. It maps its parameter to the value of the derivative of the position function at this instant. In the following we will refer to a tempo curve defined as a function of position as an *autonomous tempo curve*, whereas a tempo curve defined as a function of time will be referred to as a *non-autonomous tempo curve*. This naming stems from the formulation of the tempo curve as the right-hand side of an autonomous or non-autonomous differential equation:

$$\begin{aligned}
&\frac{dP}{dt}(t) = \boldsymbol{\mathcal{T}}(P(t)) \quad \text{(autonomous)}\,, \\
&\frac{dP}{dt}(t) = \boldsymbol{\mathcal{T}}(t) \quad \text{(non autonomous)}\,, \\
&\text{with initial condition } P(0) = 0\,.
\end{aligned} \tag{4.1}$$

## 4.3 Tempo Curves Integration

Tempo curves in Quadrant are defined as piecewise functions. For the sake of brevity, we may refer to an interval and its associated sub-function as a tempo curve *segment*, or simply as a *curve*, where the meaning should be clear from context. Each segment is defined by a start tempo and an end tempo, a duration, an interpolation mode and optional interpolation parameters. We implemented three interpolation modes, namely *constant*, *linear* and *parametric*.

### 4.3.1 Integration of Constant and Linear Tempo Curves

Constant and linear tempo curves can be solved analytically. We show below the differential equation of tempo, and the position and time functions for each case. In this section, $\mathcal{T}_0$ and $\mathcal{T}_1$ denotes the tempo, respectively at the beginning and at the end of the interval.

**Constant Tempo.**

$$\boldsymbol{\mathcal{T}}(p) = \boldsymbol{\mathcal{T}}_0 \,. \tag{4.2}$$

$$\begin{aligned} T(p) &= \frac{p}{\boldsymbol{\mathcal{T}}_0} \,, \\ P(t) &= t \times \boldsymbol{\mathcal{T}}_0 \,. \end{aligned} \tag{4.3}$$

**Autonomous Linear Tempo.**

$$\boldsymbol{\mathcal{T}}(p) = \boldsymbol{\mathcal{T}}_0 + \alpha p \,, \text{ where } \alpha = \frac{\boldsymbol{\mathcal{T}}_1 - \boldsymbol{\mathcal{T}}_0}{L} \,. \tag{4.4}$$

$$\begin{aligned} P(t) &= \frac{\boldsymbol{\mathcal{T}}_0}{\alpha}(e^{\alpha t} - 1) \,, \\ T(p) &= \frac{1}{\alpha} \log(1 + \frac{\alpha p}{\boldsymbol{\mathcal{T}}_0}) \,. \end{aligned} \tag{4.5}$$

**Non-autonomous Linear Tempo.**

$$\boldsymbol{\mathcal{T}}(t) = \boldsymbol{\mathcal{T}}_0 + \alpha t \,, \text{ where } \alpha = \frac{\boldsymbol{\mathcal{T}}_1 - \boldsymbol{\mathcal{T}}_0}{L} \,. \tag{4.6}$$

$$\begin{aligned} P(t) &= \boldsymbol{\mathcal{T}}_0 t + \frac{\alpha}{2} t^2 \,, \\ T(p) &= \frac{\sqrt{\boldsymbol{\mathcal{T}}_0^2 + 2\alpha p} - \boldsymbol{\mathcal{T}}_0}{\alpha} \,. \end{aligned} \tag{4.7}$$

### Numerical Considerations

Some of the above time and position functions are indeterminate forms for $\alpha \to 0$. To avoid that problem, we approximate these expressions by their Maclaurin series expansions in $\alpha$ when $|\alpha|$ is smaller than a given threshold. For instance, our approximation of the position function for the autonomous case when is $|\alpha| < 10^{-9}$ is:

$$P(t) \approx \boldsymbol{\mathcal{T}}_0(t + \frac{\alpha}{2}t^2 + \frac{\alpha^2}{6}t^3 + \frac{\alpha^3}{24}t^4 + \frac{\alpha^4}{120}t^5)\,. \tag{4.8}$$

## 4.3.2 Parametric tempo curves.

In this section we will give a definition of a parametric tempo curve, and show the differential equations that need to be solved in order to compute the time and position functions. These equations are then solved by a numerical solver.

An autonomous (resp. non-autonomous) parametric tempo curve segment is defined as a function $\mathscr{C}$ of the position $p$ (resp. of the time $t$), which describes the same curve in the plane $(p, \boldsymbol{\mathcal{T}})$ (resp. $(t, \boldsymbol{\mathcal{T}})$) as a parametric curve $\boldsymbol{B}(s)$ with components $B_x(s)$ and $B_y(s)$[5].

### Autonomous Parametric Tempo.

The differential equation corresponding to an autonomous tempo curve can be written as

$$\frac{dP}{dt}(t) = \mathscr{C}(P(t))\,. \tag{4.9}$$

**Position function $P(t)$.** The derivative of the position with respect to time is directly expressed by the autonomous tempo curve,

$$\frac{dP}{dt}(t) = B_y(s)\,,\ \text{where } s = B_x^{-1}(P(t))\,. \tag{4.10}$$

**Time function $T(p)$.** We operate the change of variable $s = B_x^{-1}(p)$ on Equation 4.9. Finding the time function is then a matter of solving the differential equation[6]

$$\frac{d\tilde{T}}{ds}(s) = \frac{B'_x(s)}{B_y(s)}\,,\ \text{with } \tilde{T}(s) = T(p)\,. \tag{4.11}$$

[5] Textbooks usually choose the letter $t$ to denote the parameter of parametric curves. We instead choose the letter $s$ to disambiguate it from time.

[6] Note that here we can find an equation in $s$. This is beneficial as it allows our numerical solver to find the parameter $s$ once, and then evaluate the Bézier curve and its derivatives

**Non-autonomous Parametric Tempo.**

The definition of the non-autonomous parametric tempo curves can be written as

$$\frac{dP}{dt}(t) = \mathscr{C}(t)\,. \tag{4.12}$$

**Position function $P(t)$.** Using the change of variable $s = B_x^{-1}(t)$ and the chain rule, we can write the differential equation for the position function as

$$\frac{d\tilde{P}}{ds}(s) = B_y(s)B_x'(s)\,,\ \text{with}\ \tilde{P}(s) = P(t)\,. \tag{4.13}$$

**Time function $T(p)$.** Using the formula for the derivative of inverse functions on Equation 4.12, we get

$$\frac{dT}{dp}(p) = \frac{1}{\mathscr{C}(T(p))} = \frac{1}{B_y(s)}\,,\ \text{where}\ s = B_x^{-1}(T(p))\,. \tag{4.14}$$

### 4.3.3 Bézier Tempo Curves.

The above formulation allows the use of any parametric curve, provided that it describes a derivable, non null function. Our specific implementation uses cubic Bézier curves, which are especially versatile, as they allow putting constraints on both endpoints and their first derivative, while ensuring that the curve remains contained inside its control points' convex hull. They are also intuitive to manipulate and map well to the curve-editing interfaces commonly used in animation, audio, and video applications.

An autonomous (resp. non-autonomous) Bézier tempo curve segment is defined by the parametric curve

$$\boldsymbol{B}(s) = \boldsymbol{C}_3 s^3 + \boldsymbol{C}_2 s^2 + \boldsymbol{C}_1 s + \boldsymbol{C}_0\,, \tag{4.15}$$

where the $C_i$ are the power basis coefficients computed from the Bézier curve's control points as follows:

$$\begin{aligned}
\boldsymbol{C}_0 &= P_0\,,\\
\boldsymbol{C}_1 &= -3P_0 + 3P_1\,,\\
\boldsymbol{C}_2 &= 3P_0 - 6P_1 + 3P_2\,,\\
\boldsymbol{C}_3 &= -P_0 + 3P_1 - 3P_2 + P_3\,.
\end{aligned} \tag{4.16}$$

using only polynomials in $s$. The same change of variable is done in Equation 4.13. It is unfortunately useless with Equation 4.10 and Equation 4.14, since $s$ can't be computed directly from the independant variable.

**Monotonicity**

In general, Bézier curves don't represent functions (they can have multiple points with the same abscissa, or even self-intersect), and thus can't be used to represent a tempo, which is a function of time.

To ensure that the curve describes a function, the cubic function $B_x(s)$ must be monotonically increasing on $[0,1]$. Given end points $P_0$ and $P_1$ with with abscissae $P_{0,x} < P_{3,x}$, we want a condition on the abscissae of the intermediate control points $P_{1,x}$ and $P_{2,x}$ for this property to hold.

Let $\alpha$ and $\beta$ be the respective ratios of the derivatives $B'_x(0)$ and $B'_x(1)$ at the end points over the slope of the the line segment joining $B_x(0)$ and $B_x(1)$.

Fritsch and Carlson (1980) derived the monotonicity region $\mathscr{M}$ of values $(\alpha, \beta)$ for which a cubic interpolant between two data points is monotonic (Figure 4.2). It is the union of several sub-regions defined by the following equations:

$$\begin{aligned}
&\alpha + \beta - 2 \leq 0\,, \\
&\alpha + \beta - 2 > 0 \text{ and } 2\alpha + \beta - 3 \leq 0\,, \\
&\alpha + \beta - 2 > 0 \text{ and } \alpha + 2\beta - 3 \leq 0\,, \\
&(\alpha - 1)^2 + (\alpha - 1)(\beta - 1) + (\beta - 1)^2 - 3(\alpha + \beta - 2) \leq 0\,.
\end{aligned} \tag{4.17}$$

**Remark.** *The first three regions are delimited by straight lines. The last region is the interior of the ellipse of center* $(2,2)$ *which is tangent to the coordinates axes at points* $(3,0)$ *and* $(0,3)$.

We now express $\alpha$ and $\beta$ in terms of the Bézier control points:

$$\begin{aligned}
\alpha &= \frac{B'_x(0)}{B_x(1) - B_x(0)} = \frac{3(P_{1,x} - P_{0,x})}{P_{3,x} - P_{0,x}}\,, \\
\beta &= \frac{B'_x(1)}{B_x(1) - B_x(0)} = \frac{3(P_{3,x} - P_{2,x})}{P_{3,x} - P_{0,x}}\,.
\end{aligned} \tag{4.18}$$

Without loss of generality, we can map the end points $P_{0,x}$ and $P_{3,x}$ respectively to 0 and 1. This gives us:

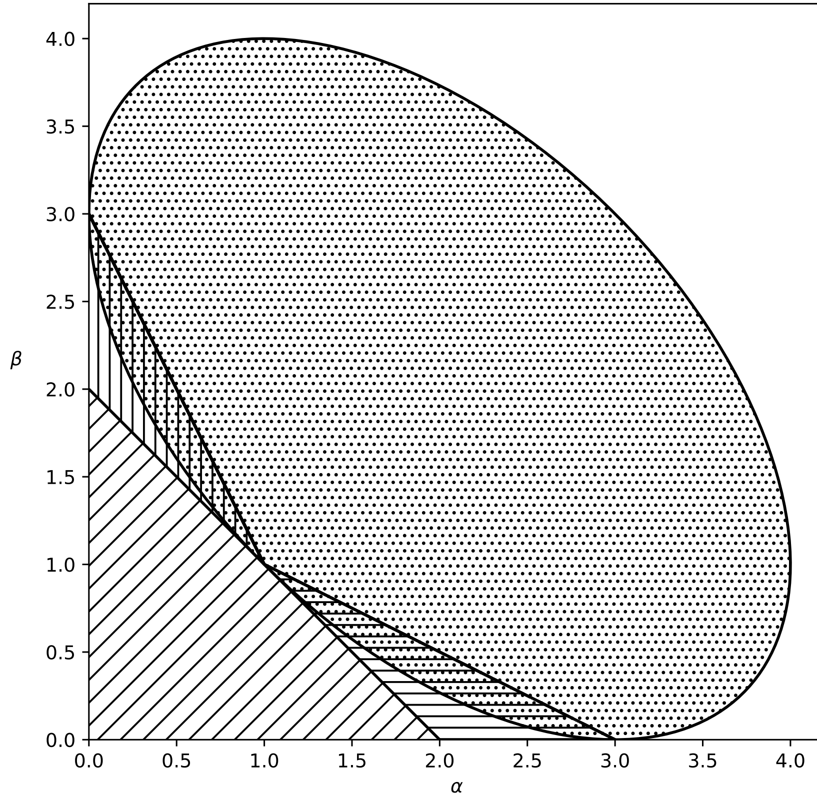



Figure 4.2: Monotonicity Region $\mathscr{M}$ (Reproduced from Fritsch and Carlson, 1980).

$$\begin{aligned} \alpha &= 3P_{1,x}\,, \\ \beta &= 3(1 - P_{2,x})\,. \end{aligned} \tag{4.19}$$

The corresponding monotonicity region for control points abscissae $P_{1,x}$ and $P_{2,x}$ is shown in Figure 4.3. Note that the unit square is entirely contained within the monotonicity region. This means that as long as the abscissae of the intermediate control points lie between the abscissae of the end

points, the Bézier curve can be used to specify a proper function. This fortunately lead to an intuitive user interface constraint: we can simply restrict the abscissae of control points to the temporal extents of the curve segment the user is editing.

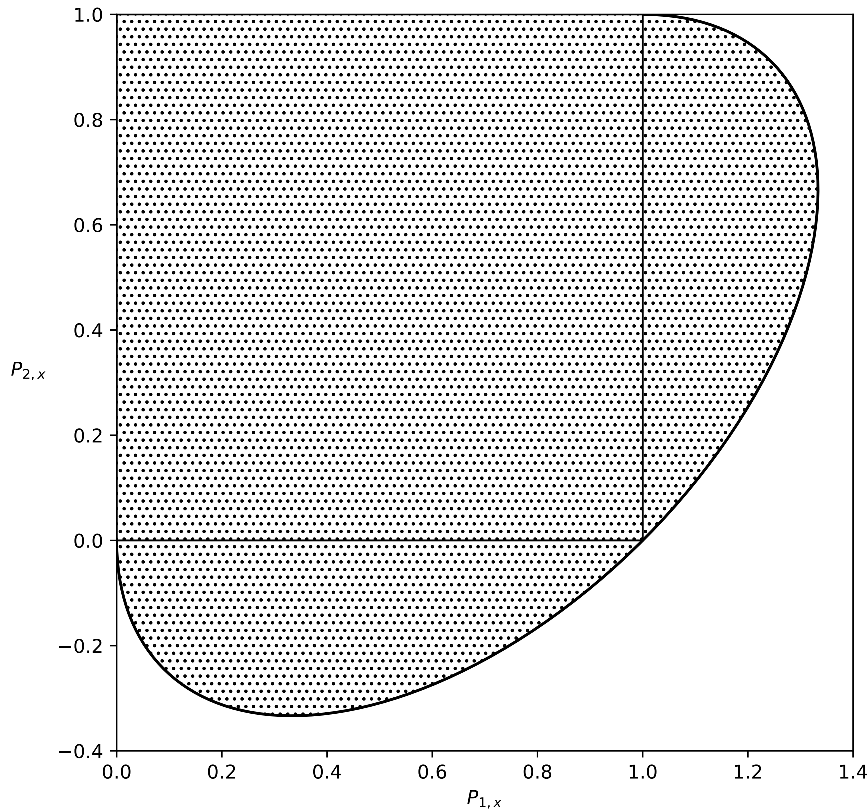



Figure 4.3: Unit square and monotonicity Region $\mathscr{M}$ for control points $(P_{1,x}, P_{2,x})$.

**Bézier curves evaluation**

We should stress out that, although each coordinate of the parametric Bézier curve is cubic with respect to its parameter $s$, the second coordinate is *not* a cubic function of the first, i.e. the tempo is not a cubic function of position

(resp. time). Analytically finding the tempo for a given position (resp. time) indeed requires solving a third order equation.

A faster method is to numerically find the parameter $s$ for a given position (resp. time), up to some desired precision, and then compute the tempo from $s$. Our implementation first uses the Newton-Raphson root-finding method up to a fixed number of iterations, and falls back to a bisection algorithm if either the value of the derivative falls behind some threshold, or the desired precision is not reached within the maximum iteration count.

### Numerical resolution

Using Bézier curves, Equation 4.13 poses no difficulty and can be analytically solved by integrating a 6th degree polynomial.

Bringing a symbolic calculus package to the rescue, one can find an analytical solution for Equation 4.11, although it involves complex logarithms and require computing the roots of the third degree polynomial $B_y$, as shown in Equation 4.20. Likewise Equation 4.10 and Equation 4.14 have no simple analytical solution.

$$\sum_{k=1}^{3} \frac{\ln(s - r_k)(C_{1,x} + 2C_{2,x} r_k + 3C_{3,x} r_k^2)}{C_{1,y} + 2C_{2,y} r_k + 3C_{3,y} r_k^2}, \quad \text{with } r_k \text{the roots of } C_{3,y}s^3 + C_{2,y}s^2 + C_{1,y}s + C_{0,y}\,. \tag{4.20}$$

It is unclear at this point if there is a really compelling reason to favor the analytical solutions. A number of corner cases would have to be considered when finding (or numerically approximating) the roots, and special care would be necessary near singularities (much as discussed in subsubsection 4.3.1).

On the other hand, using a numerical solver has the advantage of allowing us to control the tradeoff between accuracy and speed, and opens up the possibility of supporting other arbitrary functions to define tempo curves.

We use the Cash-Karp (Cash & Karp, 1990) method to numerically solve the Bézier tempo curve equations. It is an adaptive Runge-Kutta (Butcher, 1987) solver with orders 5 and 4. We follow the general architecture proposed in (Press et al., 1992), and adapt it to our needs.

Since the equations shown in subsection 4.3.2 are either autonomous or directly integrable, we wrote two specialized step routines, `autonomous_step()` and `integrate_step()`, that avoid much of the computation involved in a general Cash-Karp step. Given the value of the solution at the previous step, and a function pointer to compute the right-hand side of the equation, these routines advance the solution over a single step of given size, and return the result along with an estimation of the local error.

These routines are used in controlled-step routines which either accept the step, or adapt the step size and retry the step, depending on the local error estimate. These controlled-step routines are in turn called by driver routines that perform successive controlled steps across the desired interval. The drivers also adapt the error criterion at each step to the amplitude of the computed solution and its derivative, in order to achieve constant fractional errors.

### 4.3.4 Multi-segment curves implementation

Time transformation curves are created from a structure called a *curve descriptor*. This descriptor specifies the kind of curve (i.e. autonomous or non-autonomous tempo, or time map), and a list of *curve segments descriptors*.

A *curve segment descriptor* is a structure containing the duration of the segment, the start value and end value, the interpolation mode and its optional control parameters.

At creation time, the list of segments descriptor is processed to produce a list of internal segments. These segments contain the precomputed values of the slope $\alpha$ of linear segments or the power basis coefficients $C_i$ of Bézier segments. The value of the dependant variable (i.e. $t$ for an autonomous tempo curve, or $p$ for a non-autonomous tempo curve) at the end of the interval is also precomputed and stored in the tempo segment at creation time.

Queries on tempo curves are then handled as follows:

- The input position (resp. time) is checked against the precomputed positions (resp. times) at the breakpoints, in order to find the segment within which the input parameter lies.
- If no segment is found, the tempo is curve is conceptually extended with constant tempos on the left and on the right, corresponding to

the start tempo of the first segment, and the end tempo of the last segment.

- The output time (resp. position) is computed by solving the corresponding differential equation on the selected curve segment, using closed-form expression for constant and linear segments, or the numerical solver for Bézier segments.

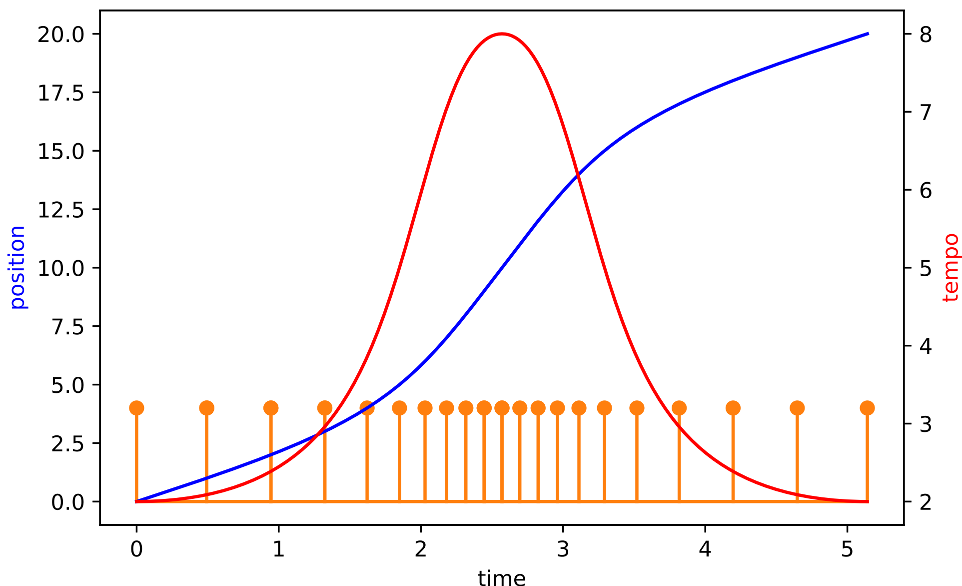



Figure 4.4: Time map and beats trace for a tempo curve defined by two Bézier curves.

An example of a time transformation produced by a tempo curve composed of two Bézier segments is shown in Figure 4.4. The blue curve shows position as a function of time. The orange stems mark the timeline symbolic time units. The red curve shows the tempo curve, as a function of time. The figure is produced by computing the positions corresponding to a regularly spaced time grid.

## 4.4 Phase Synchronization

So far we only considered synchronizing relative clock frequencies, which is more formally called *syntonization*. However, when dealing with ensemble

music, the notion of synchronization is really about the relative phase of each musician. It is also important to note that musical speed and phase are understood in the context of musical structure.

The musical structuration of time, at a very basic level, usually consists in a *meter*. Loosely speaking, a meter is a pattern describing a recurring grouping of pulses, and the relative lengths and accentuation of those pulses. The pulses are usually referred to as *beats*, and the groupings as *bars*. It is useful to keep in mind that not all beats are of equal nature within a bar, nor do they convey the same synchronization requirements.

To better highlight the fundamental difference between purely tempo-based synchronization and phase synchronization, let's imagine a DJ show based around samples of measured music. Although the sequencer can maintain virtually perfect tempo relations between the samples, and even presuming on the DJ's mastery, there are still many occasions where the samples will be slightly out of sync (input delay being the most obvious reason). Given enough information on the audio file, modern software such as Ableton Live (Ableton, n.d.) will already correct that asynchrony, by shifting and time-stretching the sample to align its beats on a predefined click track. They can even delay the triggering of a sample for it to happen on a bar boundary, a feature called *quantized launch* in Live's lingo.

Now, if our hypothetical show were to feature several DJs, there would be a need to synchronize multiple sequencers on shared beats and bars boundaries. Ableton Link (Goltz, 2018) allows sharing a common notion of playback transport and beat-based synchronization. Each connected software can control the transport and set its speed. Each application also chooses a beat quantum. Two applications sharing the same beat quantum value will be beat-aligned modulo that quantum. An application $A$ with a beat quantum being a multiple of another application $B$'s beat quantum will also be synchronized on $B$'s boundaries.

Link offers an elegant model to select a subset of beats of $A$ and a subset of beats of $B$ to align. It certainly has the merits of being simple, easy to grasp from a user's point of view, and well-suited to regular meters. However, this model induces some limitations:

- This model implies a shared tempo. Pieces with multiple tempos, such as Charles Ives' *The Unanswered Question*, Stockhausen's *Gruppen*, or Steve Reich's *Piano Phase*, would be impractical to author and perform in such a model.

- Polyrhythms can also benefit from the ability to use several pulses, instead of the lowest common multiple of all groupings, which rapidly becomes difficult. For instance, exactly synchronizing a quintuplet over a septuplet would require counting 35 beats in this model, which would incur the use of an impractically high tempo.
- This model implies beats of equal lengths. Thus it is difficult to express additive meters without resorting, once again, to a common subdivision.
- The notion of a beat quantum implies that the only beats considered for boundary alignment are congruent to 0 modulo the beat quantum. In other word, alignment always happen at the beginning of the recurring group. But let's imagine a simple scenario, in which we have a kick sample $K$ which must be aligned on 4 beat bars. Then we have a sample $S$ of on-beats snare hits, that we want to shift by one beat in order to align it on the off-beats of $K$. This would call for the possibility of specifying arbitrary remainders in addition to the beat quantum: in our minimal example, the on-beats of $S$ must be aligned to beats of $K$ which are congruent to 1 or 3 modulo 4. One can certainly contemplate much more intricate scenarios, which would need manually specifying desired synchronization points altogether.

More generally, we think a meter-centric alignment model like Link's shouldn't be conflated with the more general notion of date alignment. In particular, the primitive of the system shouldn't be built around the assumption of congruent beat quanta, lest it be hampered by the shortcomings discussed above. Instead, at the most basic level, the synchronization system should only be concerned with aligning a future date and tempo of a process $A$ to a a given target, in a given amount of time. This goal can be stated as follow:

Given a processes $A$ and date $t_0$ in its parent's timeframe, corresponding to position $p_0$ in $A$'s local timeframe; Given $\boldsymbol{\mathcal{T}}_0$ the tempo of $A$ at $t_0$; We want to apply a time transformation to $A$ such that at date $t_1 = t_0 + \delta$, $A$ reaches a given position $p_1$ and a given tempo $\boldsymbol{\mathcal{T}}_1$.

The target $(t_1, p_1, \boldsymbol{\mathcal{T}}_1)$ can then be chosen to align some beats of $A$ onto some beats of another measured process $B$. It can also be used to track a variably paced process $B$, such as the output of a score follower inferring position and tempo values from a human player. This is similar to the dynamic *target* synchronization strategy used in Antescofo (J.-M. Echeveste,

2015), although we use a different catch-up curve. The target can also be used to smear the timing of $A$ based on irregular or punctual commands, not necessarily tied to a listening machine, such as *"skip two beats over the next 10 seconds"*. Note that this doesn't precludes the use of a simpler, meter-centric interface built *on top* of this basic capability.

### 4.4.1 Catch-Up Curves

Our scheduler itself doesn't directly handle date alignment. Instead, this is achieved by computing a catch-up time map curve and applying it to the task to be aligned. We want the time map to satisfy the following constraints:

$$\begin{aligned} \mathscr{C}(t_0) &= P_0\,, \\ \mathscr{C}'(t_0) &= \boldsymbol{\mathcal{T}}_0\,, \\ \mathscr{C}(t_1) &= P_1\,, \\ \mathscr{C}'(t_1) &= \boldsymbol{\mathcal{T}}_1\,. \end{aligned} \tag{4.21}$$

This ensure $\mathscr{C}$ is a smooth transition without sudden jumps or accelerations. Fortunately, Bézier curves allows us to control both endpoints and derivatives. The constraints implies the following equations on the Bézier curve control points:

$$\begin{aligned} P_0 &= (t_0, p_0)\,, \\ P_1 &= (t_0 + \delta_1, p_0 + \boldsymbol{\mathcal{T}}_0\delta_1)\,, \\ P_2 &= (t_1 - \delta_2, p_1 - \boldsymbol{\mathcal{T}}_1\delta_2)\,, \\ P_3 &= (t_1, p_1)\,, \end{aligned} \tag{4.22}$$

**Remark.** *This means that $P_1$ (resp. $P_2$) must lie on the tangent to $\mathscr{C}$ at $P_0$ (resp. $P_3$).*

Furthermore, the Bézier curve must define a monotonically increasing function. This means both $B_x(s)$ and $B_y(s)$ cubics must be monotonic. This can readily be enforced by applying the constraints shown in subsection 4.3.3, but this time both on the abscissae and ordinates of the intermediate control points $P_1$ and $P_2$.

## 4.5 Scheduler

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 (Halbwachs, 1993), 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 Von Hanxleden et al. (2017), 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.

### API design discussion

The API of a scheduler essentially determines how a user will feed code fragments into the scheduler, to be executed later at a specified time. There are several important parameters to consider here: what constitutes a code fragment? How is a piece of data passed along with a fragment that operates

on it? Are these fragments executed in the same context[7] as the code that provided them to the scheduler?

A common approach to that problem is to expose a callback mechanism, where the user can register functions to be called by the scheduler at a later date. This is the approach adopted, among others, by Max clock API[8], by the musical objects scheduler developed for OpenMusic in (Bouche & Bresson, 2015), or in a number of Web APIs such as (Schnell et al., 2015) and (Roberts et al., 2017). In this model, the unit of code that can be provided to the scheduler is a function, and is usually executed in a different context that the code that registered it. Thus, the registering mechanism must also provide a way to capture some data for the callback.

Although this approach is relatively straightforward and well suited to the scheduling of independent actions, it has a number of shortcomings. In particular, when it comes to organizing streams of related actions in time, the callback model compels the user to break down the control flow of its code into lots of little distinct functions. Not only does it obfuscate the logical relationship of actions as well as their sequentiality, but it also puts on the user the unnecessary burden of passing a shared context around, ensuring its consistency and managing its lifetime. Sequential streams of actions would be better expressed by sequential code executed within the same context.

Another approach is to design the scheduler to run user code in fibers[9]. In this approach, the fragments of code provided to the scheduler by the user are not constrained to be self-contained callbacks, and sequential or logically related actions can be grouped within a straightforward control flow, and share the same local context.

Fibers fundamentally differ from threads in that they implement a cooperative scheduling model: the scheduler API exposes functions to explicitly yield and reschedule the calling code to a future date. In their simplest form, they provide concurrency, but not parallelism. This is what we need in our

[7]The word *context* here is left intentionally broad. It could encompass the threading model, the data lifetime, the nature of the execution environment (e.g. native code scheduling interpreted code and vice versa), or even the machine on which the code is executed.

[8]See https://cycling74.com/sdk/max-sdk-7.3.3/html/chapter_scheduler.html

[9]The notions of *fiber*, *coroutine*, or *green thread* are very closely related, and the distinction between them, if any, is amenable to debate. One could argue that *green thread* is more appropriate in the context of a virtual machine or runtime environment, while *coroutine* originates from programming language design. The term *fiber* may capture a more general view of the concept, but more importantly it happens to be shorter to type.

case, since we want to interleave computations in a deterministic way, under the control of user code[10].

A fiber-based scheduling API can feature mechanisms to wait for other fibers to complete, thus allowing to express dependencies between different workloads and execute them in the correct order, something which is impractical to implement with a callback API as it would require the construction of an explicit dependency graph beforehand. Another advantage of this model is that a fiber can easily be migrated between thread, which allows a very streamlined way to handle blocking calls without hanging the scheduler. Finally, callback semantics are readily emulated by spawning a new fiber that doesn't yield until it terminates. Hence, if considered strictly from the point of view of the usage code, this API design is strictly superior to the callback-based one. Its cost is a slightly higher complexity on the implementation side, which has to maintain and swap fiber contexts as needed. Another downside of fibers is the weaker support for visualization and line stepping of fiber code in most debuggers. That said, a callback-based API is also hard to debug due to the disrupted control flow and the non-obvious sharing of data across callbacks.

### 4.5.1 Scheduler API

Since our objective is to provide ways to specify and organize highly interdependent computations into complex temporal scenarios, we decided to base our scheduler architecture on the fiber approach. This is also the type of architecture adopted by FORMULA and Antescofo. However, FORMULA constitutes a whole operating system and its fibers are really more akin to system-level cooperative processes, while Antescofo is an interpreted language whose fibers are implemented at the interpreter level. On the other hand, Quadrant features user-level, native code fibers.

The API declaration is summarized in (Listing 4.1). The scheduler system is initialized by a call to `sched_init()` and shut down with `sched_end()`. User code can then interact with the scheduler by operating on *fiber* handles. A fiber represents the execution context of a piece of user code associated with a given timeframe. Fibers are organized in a parent-children relationship.

[10]Doing this with thread is of course possible, but ill-advised. Indeed, we would have to pay the cost of synchronization primitives to essentially *undo* the parallelism and preemptive scheduling properties of threads.

```
// Initialization and shutdown
void sched_init();
void sched_end();

// fiber creation
sched_fiber sched_fiber_create(sched_fiber_proc proc, void* userPointer);
sched_fiber sched_fiber_create_detached(sched_fiber_proc proc,
                                        void* userPointer);

sched_fiber sched_fiber_create_for_parent(sched_fiber parent,
                                          sched_fiber_proc proc,
                                          void* userPointer);

// fibers' timescales
void sched_fiber_timescale_set_scaling(sched_fiber fiber, f64 scaling);
void sched_fiber_timescale_set_tempo_curve(sched_fiber fiber,
                                           sched_curve_descriptor* desc);

// Scheduling
void sched_wait(sched_steps steps);
void sched_suspend();
void sched_cancel();

void sched_fiber_suspend(sched_fiber fiber);
void sched_fiber_cancel(sched_fiber fiber);
void sched_fiber_resume(sched_fiber fiber);

sched_wakeup_code sched_wait_for_task(sched_object_handle handle,
                                      sched_object_signal signal,
                                      sched_steps timeout);
// Handles management
void sched_handle_release(sched_object_handle handle);
sched_object_handle sched_handle_duplicate(sched_object_handle handle);

// Background jobs
void sched_background();
void sched_foreground();
```

Listing 4.1: Quadrant's scheduler internal API.

**Fiber creation.** User code can create and schedule a new fiber by calling one of the `sched_fiber_create_XXX()` functions. The default version creates a new fiber as a children of the current fiber. The `detached` version creates a new fiber synchronized to the clock-time. The third version creates a new fiber as a children of another fiber. Once the fiber is picked by the scheduler to be run, it will start executing the entry procedure passed in the `proc` parameter, passing it the value of `userPointer`. The fiber creation

functions return a fiber handle, which can subsequently be used to operate on the fiber or query its properties.

**Time transformations.** The timescale associated with a fiber can be configured to apply a specified transformation to its source time, by using one of the `sched_fiber_timescale_set_XXX()` functions. The transformation can be specified as a simple tempo scaling or as a tempo curve (see section 4.2).

**Wait, suspend, resume, cancel.** At any point in the execution of a fiber, user code can call `sched_wait()` to yield to the scheduler and reschedule the fiber in `steps` units of time. A call to `sched_suspend()` suspends the calling fiber, which can be resumed by a call to `sched_fiber_resume()`. A call to the function `sched_fiber_suspend()` suspends a fiber, which means that the timescale associated with the fiber is no longer updated and its playhead stops advancing. A suspended fiber can be resumed by calling the function `sched_fiber_resume()`. A fiber can be canceled by a call to `sched_fiber_cancel()`.

**Fibers lifecycles.** Fibers can be running, suspended, retired, or completed. Figure 4.5 shows a how fibers transition between running, suspended, retired and terminated states. A fiber is running until it returns from its entry procedure or it is canceled, at which point it will be marked as *retired*. If it has no children, or all its children are completed, it will be marked as completed. Most resources associated with fibers are recycled as part of the retirement and completion stages, but a minimal set of resources is kept alive until all handles to the fiber are closed by a call to `sched_fiber_release()`. This is done so that handles can still be queried for some properties (such as status or exit codes) after the object they reference is completed. Handles can be duplicated by a call to `sched_handle_duplicate()`, should the object's lifetime be dynamically extended across multiple use-sites.

**Waiting on handles.** A fiber can wait until another fiber is retired or completed, by calling one of the waiting functions on its handle. The most generic version waits on a handle for a specified `signal` (i.e. retirement or completion), or until a specified `timeout` has elapsed. All these functions return a wakeup code, which indicates the condition on which the wait was ended.

**Background jobs.** A fiber should never hang the scheduler and prevent other fibers from progressing by failing to yield in a timely fashion. In fact we have plans for a watchdog mechanism that will interrupt and cancel such

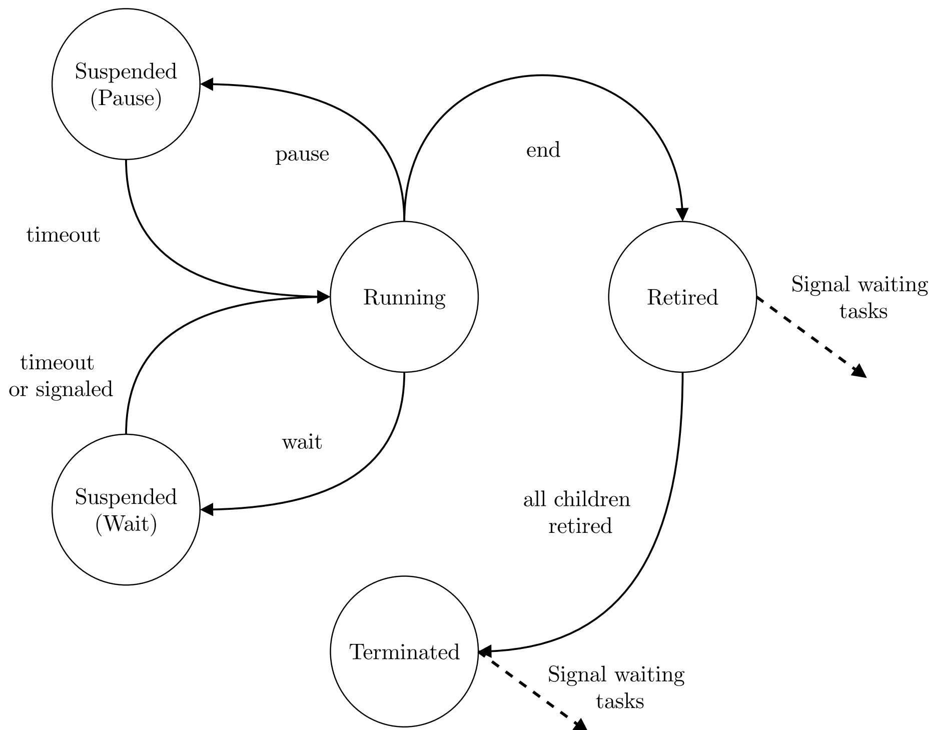



Figure 4.5: Scheduler fibers state diagram

ill-behaved fibers. In order to comply with this rule, fibers that must call a blocking routine or perform untimed, lengthy computations, should request to be put on a background thread by calling `sched_background()`. Once their blocking work is done, they can reintegrate the normal scheduling flow by making a call to `sched_foreground()`.

### 4.5.2 Scheduler's operation

The functions of the scheduler and the integration of tempo curves are respectively defined in the `scheduler.cpp`[11] and `sched_curves.cpp`[12] files,

[11] https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/scheduler.cpp

[12] https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/sched_curves.cpp

and their corresponding header files. The native fiber switching logic is implemented by a few lines of inline `x64` assembly in `x64_sysv_fibers.cpp`[13].

### Main loop

Fibers are organized as a collection of trees corresponding to the synchronization relationships of their timescales: root fibers are synchronized on the real-time clock, and every other fiber is synchronized on its parent timeframe. The scheduler also maintains a list of all running fibers, each associated with their date of next execution.

The scheduler's role is to execute actions at the clock-time corresponding to their logical date. In order to do so it must maintain the timescales's playheads consistent with the clock-time, through the time transformations, pick the next fiber to execute, sleep until its real time due date is reached, and execute it. It also has to handle control messages that may interrupt a sleep. The main scheduler algorithm, implemented by the `sched_run()` procedure, is a loop whose general outline is given in Listing 4.2.

There is a subtlety in how the playheads are advanced: if the sleep was interrupted by an external message, the effective duration of the sleep `timeSlept` is used to advance the playhead. But if the process wakes-up due to the sleep's timeout, the duration used to advance the playheads is the logical duration of the sleep, i.e. the value of `nextDelay`. This could well be different from the effective duration of the sleep, but the scheduler maintains the illusion of a perfectly accurate sleep timing. This is done so to avoid accumulating errors when action results in the scheduling of new events, relative to the current date. To preserve the absolute timing accuracy, the scheduler maintains a accumulator of `nextDelay - timeSlept`, which is added to the timeout just before passing it to the sleep call.

Another measure that we take in order to improve timing accuracy is what we call *Zeno sleeping*. It stems from the observation that on our target operating systems, the accuracy of waiting calls is somewhat proportional to the timeout parameter. Thus, we wrap our sleep calls in a loop, with each iteration using a timeout equal to a fraction of the timeout residue from the previous iteration. The loop exits when the residue falls below the desired accuracy (or if the sleep duration is too small to be measured on the system). This allows us to achieve sub-millisecond accuracies with very few

[13]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/x64_sysv_fibers.cpp

```
START:

FOR EACH timescale:
        Translate the date of the first event from logical time to
        clock-time. Set nextFiber to point to the event with the
        earliest date.

Store the distance between the date of the selected event and the current
clock-time in fiberDelay.

Compute how much time we can sleep before the next event.

Sleep during the amount of time computed above. The sleep can be
interrupted by an external message, e.g. to schedule new events.

Wakeup from the sleep. Advance the playheads over the duration slept,
applying the necessary time transformations.

IF the sleep was interrupted by a message:
        Handle this message.
ELSE:
        remove nextFiber from the queue and switch execution to this fiber.

Go back to START.
```

Listing 4.2: Scheduler algorithm outline

loop iterations even for long sleeps. Better accuracies can be achieved, but it is of course a tradeoff between desired accuracy and CPU usage.

### Waiting operations

A fiber whose user code calls `sched_wait()` simply yields back to the scheduler, and is put back into list of running fibers according to the duration passed to the waiting call.

Besides maintaining a list of running fibers, the scheduler also has a list of suspended fibers. When a fiber is suspended, it is simply moved from the running list to the suspended list. If the calling fiber happens to be the one that is suspended in the process, it also yields back to the scheduler. Resuming is simply a matter of moving the object back to the running list.

A fiber whose user code calls one of the “wait” functions is also moved to the suspended list. Additionally, the expected signal is stored inside the fiber structure, and the fiber is added to a list of waiting fibers inside the waited

object. When a fibers is retired or completed, its list of waiting fibers is traversed and all fibers whose expected signal matches the current operation are put back to the running list, with their wakeup code set to the constant `SCHED_WAKEUP_SIGNALED`.

A special case is fibers which are waiting on a handle with a timeout. These are put in the running list like normal running fibers, but a status flag in the fibers structure indicates that it is in a `waiting` state. If the object is signaled before the timeout elapses, the status is simply set back to `running`. On the other hand, if the fiber is picked by the scheduler to be run before its waited object is signaled, it is first removed from the waiting list of said object, its status is set back to `running`, and its wakeup code is set to the constant `SCHED_WAKEUP_TIMEOUT`.

### Background jobs

The scheduler uses a thread pool to handle background jobs. The thread pool consists of a queue of fibers, a mutex $M_{bg}$ and a condition variable $C_{bg}$, and a number of worker threads. In order for the job system to pass fibers back to the main scheduler thread, the scheduler also has a message queue protected by a mutex $M_m$ and a condition variable $C_m$. Sleeping in the main scheduler loop is implemented by waiting on $C_m$, which allows the sleep to be interrupted by the arrival of messages posted by the job system.

A call to `sched_background()` sets the status flag of the running fiber to `background`, puts it in the suspended list, and yields back to the scheduler. The scheduler then detects the status change, locks the mutex $M_{bg}$, pushes the fiber in the job queue, signals the condition variable $C_{bg}$ and unlocks the mutex.

Worker threads mostly sit idle waiting on the condition variable $C_{bg}$. When the condition variable is signaled, one worker thread wakes up, pops a fiber off the queue and switches to it. It continues executing fibers until there are no more fibers in the queue, at which points it goes back to sleep. When a background fiber calls `sched_foreground()`, it yields back to the main routine of the worker thread. This routine then locks $M_m$, puts a message containing the fiber in the message queue, signals $C_m$ and unlocks $M_m$. This message will in turn be picked by the scheduler main loop, which will put the fiber back to the running list.

### Buffered Actions

The goal of a computer music system or a multimedia show controller is ultimately to output some data to the outside world in order to provoke some action (e.g. audio samples to be converted into sound, or commands sent to external synthesizers, etc.). The operators are most concerned by the precise timing of the perceptible effects. However the system allows specifying the dates at which computations must happen, which is only a proxy for specifying the dates of actions: delay and jitter are induced by computations leading to the output. As such, it is useful to distinguish code that almost immediately result in an action (such as outputting an OSC packet), from code that actually computes the data to be sent out (such as computing the OSC parameters and formatting them into a well-formed message). The first category should always be strictly scheduled with respect to real time. Ideally, it should even be latency compensated to correct the delay between each individual system's output and its real-world effect. The second category can (and should) be scheduled as soon as feasible, in order to avoid delaying the actions that depend on their result.

The ability to make that distinction in the timing specification of the system is what FORMULA calls *action buffering* (Anderson & Kuivila, 1990). Processes in the system are executed ahead of time, and generate actions that are buffered and only executed when the real time clock reaches their due date. This prevents the timing variability of normal computation to create hiccups and jitter in the timed sequence of actions, because a computation running a little longer than usual will still be done in time to generate the desired action before its due date. However, the difference between the real time clock and the ahead-of-time clock induces an input delay, since inputs to the system can only modify the course of actions after the current ahead-of-time date. Thus there is a tradeoff between the stability of the output timings, and the reactivity of the system.

We explored the potential of action buffering in Quadrant by using a root logical clock that ran as fast as possible within a fixed look-ahead real-time window. Actions generated by internal computations were placed in a special real time queue and served at the specified real-time date. Sleeps only occur when the next event is outside the look-ahead window. The resulting scheduler loop is given in Listing 4.3 (compare with the simpler loop of Listing 4.2).

```
START:
Get the next action and its delay from the action queue.
Store them in nextAction and actionDelay.

FOR EACH timescale:
    Translate the date of the first event from logical time to clock-time.
    Set nextFiber to point to the event with the earliest date.

Store the distance between the date of the selected event and the current
clock-time in fiberDelay.

Set nextEvent to nextFiber or nextAction, whichever comes first.

Compute how much time we can sleep before nextEvent.

IF nextEvent is a fiber:
    Increase the look-ahead to reach the fiber date. We only need to sleep
    if fiberDelay is outside the maximum look-ahead window.
ELSE:
    nextEvent is an action, we need to sleep until we reach the action
    real time due date.

Sleep during the amount of time computed above. The sleep can be
interrupted by an external message, e.g. to schedule new events.

Wakeup from the sleep. Advance the playheads over the duration slept,
applying the necessary time transformations. Decrease next action delay
by the time slept. Update the look-ahead depending on the time slept.

IF the sleep was interrupted by a message:
    Handle this message.
ELSE:
    IF nextEvent is an action:
        Remove it from the queue and execute its callback.
    ELSE:
        nextEvent is a fiber, remove it from the queue and switch
        execution to this fiber.

Go back to START.
```

Listing 4.3: Scheduler algorithm with buffered actions.

Although Quadrant's current implementation only schedules code within a single OS process, it is desirable to allow future synchronization of user code across multiple processes and machines, effectively implementing a form of distributed scheduling. In light of this objective, an additional nuance has to be made regarding the nature of outputs: some messages represent real outputs reaching outside the scheduler system, while others might be

sent to remote parts of the (distributed) scheduler in order to influence its operations. The first category should be handled as before. However the second category shouldn't be buffered, and should in fact be timestamped with the logical, ahead-of-time clock, and sent as soon as possible.

A difficult problem arises when such an internal message is received by a local system whose ahead-of-time clock is already past the timestamp of the message. On the one hand, the system can overwrite the message's timestamp with the current date, at the cost of some internal inconsistency that might render the scheduling much less predictable. On the other hand, the system could prevent this situation from ever happening by mandating such interacting sub-systems to always work in lock-step, but this could create unnecessarily long latency chains, which would defeat the purpose of action buffering. A third avenue to attack this problem, would be to implement a form of selective backtracking when an old message that would have changed the flow of the local scheduler is received.

Regardless of the possible solutions to the above problems, their common flaw is to lock tradeoffs at a coarse grain level, without knowledge of the "semantics" of scheduled events. However, it is not unusual that within the same scenario, several strategies must be applied to deal with latency and jitter depending on the intended effects of the events being generated. Some events might need to be processed with very low latency, whereas some might need tight synchronization. Some might rely on strict ordering whereas some might be handled on a first-come first serve basis. Some might be ignored altogether as soon as a new event comes in and over-rules previous events. It might also well be the case that in most situations, where the granularity of computational tasks is less than the accepted timing inaccuracy threshold, buffering is completely unnecessary.

As such, our opinion is that dealing with latency and jitter is much better left to ad-hoc, context-aware solutions. For that reason, we removed buffered actions from Quadrant's scheduler and stuck with the run loop shown in Listing 4.2.

# Chapter 5

# Quadrant's Temporal Language

Quadrant provides a custom language to specify temporal scenarios[1]. As discussed in section 3.2, it is a non-textual language: although programs source representations are mostly *displayed* as text, they are in fact trees of cells containing 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 Quadrant has very different characteristics than typical Lisp dialects. Indeed, Quadrant is an imperative, statically typed language with (mostly) unmanaged memory. It has built-in cooperative concurrency and temporal primitives based on fibers. 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.

We first describe the basic programming constructs of the language (section 5.1), before describing its temporal features (section 5.2).

[1]Throughout the remainder of this work, we may simply refer to it as "Quadrant", as it should be clear from context if we are referring to the language or to the environment as a whole.

## 5.1 Basic Constructs

### 5.1.1 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)))
```

### 5.1.2 Control Flow

Quadrant has the usual basic control flow constructs like **`if`** conditionals, **`for`** and **`while`** loops, and lexical scoping with a **`do`** form:

```
(if condition
    branchIfTrue
    branchIfFalse)

(for initStatement
     conditionExpression
     iterationExpression
   body)

(while condition
   body)

(do
   body)
```

### 5.1.3 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. Quadrant predefines a number of primitive named types:

- A empty type, **`void`**.
- Sized unsigned integers **`u8`**, **`u16`**, **`u32`**, **`u64`**.
- Sized signed integers **`i8`**, **`i16`**, **`i32`**, **`i64`**.
- Floating point numbers **`f32`** and **`f64`**.
- Boolean **`b8`**.
- An boxed type, **`any`**.

A type specification can be a named type or a compound type specification using one of the following type constructors: **`array`**, **`slice`**, **`struct`**, **`ptr`**, or **`future`**.

### 5.1.4 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)
```

Slices can be made from an array or another slice by specifying an index range with an inclusive low bound and an exclusive high bound:

```
(slice low high arrayOrSlice)
```

Array and slices elements can be accessed by index with the **at** form:

```
(at index arrayOrSlice)
```

The element count of an array or slice can be accessed with the **len** form:

```
(len arrayOrSlice)
```

### 5.1.5 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
        ...)
```

### 5.1.6 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 (e.g. a variable, a structure field, or an array element). The **unref** form allows accessing a pointer's underlying value:

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

### 5.1.7 Conversions

Literal numeric values are either integers or floating point and use the largest representation (i.e. `u64` or `f64`). Literal arithmetic expressions (i.e. expression only involving literal values) can be implicitly cast in the following cases:

- The destination type is a size variant of the source type.
- The source type is an integer type and the destination type is a floating-point type.

Numeric values are implicitly converted to boolean in some constructs (such as conditions of branches and loops), where the 0 value is converted to `false` and non-zero values are converted to `true`.

Other values and types require an explicit cast:

```
(cast typeSpec expr)
```

This converts the operand `expr` to type `typeSpec`. The following conversions are valid:

- Casting numeric values to other numeric values. Downcasts (from larger to smaller variant) may overflow. Casts between same size signed and unsigned values keep the bit-level representation unchanged (i.e. values may wrap around). Floating point to integer casts are truncating.
- Casting between different pointer types. The pointed address remains unchanged.

The type of arithmetic operations is determined by the type of their operands. If operands are not of the same type, and one is literal, the checker attempts to perform an implicit cast of the literal operand to the other operand's type. If both operands are literal and both are integers or both are floating point, the checker attempts to perform an implicit cast to the largest variant one. If one is an integer and one is a floating point, the conversion is performed on the integer operand towards the floating point type.

### 5.1.8 Any Type

The type specification `any` can be used for boxed values. Any value can be implicitly cast to a value of type `any`. This effectively creates a boxed copy

of the cast value. The compiler generate runtime type information for the effective type of the `any` value, which can be queried with an internal API. This is essentially used to pass boxed values to foreign code that needs this type information, such as type-generic logging routines.

### 5.1.9 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)
```

The type of `val` must be consistent with the return type specification of the procedure definition.

A procedure is called using its name followed by arguments:

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

### 5.1.10 Variadic Procedures

Typed variadic procedures can be defined using the `varg` parameter specifier:

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

Inside the body of the procedure, `paramN` is typed as a slice of type `typeSpecN`. At call sites, the parameter `paramN` can be passed 0 or more

arguments of type `typeSpecN`. These arguments are collected into a slice that is passed to the procedure.

### 5.1.11 Polymorphic Procedures

Quadrant supports generic procedures, i.e. procedures that can be applied to different types[2]. Generic types are denoted by polymorphic identifier, which are identifiers prefixed by the `$` character. When a polymorphic procedure is called, polymorphic identifiers are assigned a concrete type, as we detail below. For each polymorphic procedure, a unique variant is instantiated for each set of concrete types with which it is called. Call sites are redirected to the correctly typed variant.

#### Implicit Polymorphism

Procedures can have polymorphic parameter types. A procedure parameter has a polymorphic type if its type specification contains one or more polymorphic identifiers. Polymorphic identifiers declared in the procedure parameter list can then be used in the return type specification and in the procedure's body.

```
(def sort (s (slice $T) -> (slice $T))
        // sort s using elements of type $T
        ...)
```

Call arguments' types are matched against polymorphic parameter types, and match results are assigned to the polymorphic identifiers. For instance, calling the above procedure it with an argument `s` of type `i32` would fail as `i32` isn't a slice type. Called it with an argument `s` of type `(slice i32)` would succeed and assign type `i32` to the polymorphic identifier `$T`.

#### Explicit Polymorphism

Procedures can also have explicit type parameters, which are parameters whose name is a polymorphic identifier and whose type specification is the keyword `typeid`. Type parameter names can then be used in the return type specification and in the procedure's body.

[2]Variadic and Polymorphic procedures are also covered in this video: https://youtu.be/AmO9hczGkYU.

```
(def alloc ($T typeid -> (ptr $T))
        // allocate an element of type $T
        ...)
```

When calling the procedure, a type specification must be explicitly passed to each type parameter. This type specification is then assigned to the type parameter polymorphic identifier. For instance, one can the above procedure to allocate memory for value of type `FooType` and store the result in a pointer `foo` as follows:

```
(var foo (ptr FooType) (alloc FooType))
```

This would assign type `FooType` to polymorphic identifier `$T` for this instantiation of procedure `foo`.

### 5.1.12 Module System

A Quadrant program can consist of multiple modules[3] that can reference each other's definitions. During checking, each module has a checking context, which consists of the symbols it can access. By default, the checking context is populated with the symbols defined in this module. The import directive can be used to imports the top-level symbols of another module into the local checking context.

```
(import moduleName)
```

`moduleName` is an identifier designating the module to import. Quadrant will try to find a file named `moduleName.ql`, load that file and check it.

Users select a *main* module, where the checking process begins. Imported modules are checked before the local definitions of the importing module, in the order of appearance of import directives. Modules are checked once and cached in case they're imported multiple times. Cyclic imports are forbidden and result in a checking error.

[3]Modules currently map to single files, but could be extended to map to several files.

#### Name Spaces

Symbols are imported under a given *name space*, which is by default the name of the imported module. These symbols can then be accessed in the importing module by prefixing them with the name space followed by a colon.

```
(moduleName:someProcedure x y z)
```

The name space can be changed using the `@(as)` attribute as follow:

```
(import moduleName @(as nameSpace))
```

#### Import paths

When importing a module Quadrant first looks for the module file in the same directory as the main module. If the imported module is not found there, it looks for the file in a list of predefined global search paths.

### 5.1.13 Foreign Blocks

A foreign block is used to import procedures from a foreign library (e.g. written in C). It uses the following form:

```
(foreign "Name"
   proc1
   proc2
   ...)
```

The foreign block imports the library from the file named `libName.dylib` and loads each of the declared procedures `proc1`, `proc2`, etc. from the library. Each foreign procedure declaration in a foreign block has the form:

```
(def procName (param1 typeSpec1
               param2 typeSpec2
               ...))
```

The attribute `@(linkname symbol)` can be used after a foreign procedure name to use a different symbol than `procName` to lookup the procedure in the library.

Quadrant uses the same search paths as for modules to look for the library file. The special string `"_runtime_"` can be used to import symbols from the Quadrant executable itself. This is used to import internal Quadrant APIs to interact with the virtual machine, such as the memory allocation routines or the synchronization APIs.

## 5.2 Temporal Features

Quadrant provides dedicated temporal constructs and built-in instructions to create and manage concurrent tasks and control their scheduling according to the temporal model described in chapter 4.

### 5.2.1 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 symbolic timeframe of the current task.

### 5.2.2 Standby

The `(`**`standby`**`)` instruction suspends the execution of the current task and informs the editor to show a pulsing standby icon in front of the instruction and its callers. The user can put the cursor in front of a standby instruction and hit a keyboard shortcut to resume the suspended task.

### 5.2.3 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)` attribute can be used right after the **flow** keyword to attach a time transformation to the flow. The attribute's parameter should be a `#tempo_curve` token[4]. The tempo curve is then used to map the timeframe of the task running the **flow** block statements to and from its parent task's timeframe.

```
(flow @(tempo #tempo_curve)
      ...)
```

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** statements. 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 calling task is then resumed and the **wait** form evaluates to the value returned by the **flow** 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 is true if the **flow** block has returned.

```
(timeout someFuture maxDuration)
```

---

[4] Although this is currently the only accepted parameter to the tempo attribute, this construct makes provision for passing other expressions as tempo specifications (e.g. a scalar tempo or a curve variable).

**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 alive after the task has terminated, for instance to retrieve the returned value. **wait** and **timeout** forms automatically decrement that reference count if a result is returned. However, programmers can explicitly manage task's lifetimes by using the **fdup**, which creates of copy of a future and increments the reference count of the underlying task, and the **fdrop** form, which decrements the reference count.

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

### 5.2.4 Background Pool

The `(background)` form pulls the current task out of the temporal scheduling pool and puts it in a background thread pool. It continues to execute the children statements of the background block without blocking other tasks, until it reaches the end of the block, at which point the task is moved back in the temporal scheduling pool. It is an error to call other temporal primitives while in the background pool.

### 5.2.5 Phase Synchronization

The phase synchronization features (section 4.4) can be accessed through the `sync` module. This module exposes procedures to get the current task's position, parent's time, and tempo, and allows tracking an external synchronization source using the `sync:beat` procedure:

```
(import sync)
...
(sync:beat sourcePos sourceTempo)
```

This procedure steers the calling task to catch-up with a process currently executing position `sourcePos` with tempo `sourceTempo`.

The catch-up duration is currently proportional to the offset between the calling task's current position and `sourcePos`. The alignment date at the

end of the catch-up is inferred from `sourcePos` by assuming `sourceTempo` stays constant. Future finer models could infer a variable tempo, taking into account the history of past calls to `sync:beat`.

## 5.3 Example Program

This section presents a small example program that generates polyphonic random walks in a pentatonic scale to pilot a windpipes synthesizer. It was used in a presentation of Quadrant given at the Sound and Music Computing Conference 2022[5].

Figure 5.1 shows imports and helper procedures that are used by the scenario. The program first imports packages used to manage memory, serialize OSC messages and send them as a UDP packet.

The `prngState` local variable and the `prng` procedure define a simple pseudo-random number generator. It is used in the `nextNote` procedure to select a displacement from the `dispTable` slice. That displacement is used to move from the current note, described by the `state` argument, to the next note, using a pentatonic scale specified by the `scale` slice.

The `playNote` procedure generates a note using the `nextNote` procedures, builds an "note on" OSC message, and sends it to the synthesizer. It then pauses for the duration of the note, builds an "note off" OSC message and send its to the synthesizer.

Figure 5.2 shows the entry point of the program. First the `start` procedure opens a socket to send UDP messages to the synthesizer and initializes the backing memory for the OSC messages. It then creates a main `flow`, that creates two `children`, each corresponding to a voice. Each voice walks its random note sequence in a specified octave range, by repeatedly calling `playNote` in a loop. The main flow waits for the two futures of its children, and the `start` procedure waits for the main flow. The main flow also has a tempo attributes, which controls the playback tempo of both voices.

[5]The recording of this presentation can be seen here: https://www.youtube.com/watch?v=6nC2M3NwDe8.

```
(import net)
(import osc)
(import mem)
/* pseudo-random number generator */
(var prngState u64 45798)
(def prng (-> u64)
   (set prngState (% (+ (* 75 prngState) 74) 65537))
   (return prngState))

/* random note sequence generator */
(var scale (slice i32) [64 65 69 71 72])
(var dispTable (slice i32)
   [0 1 1 1 (- 1) (- 1) (- 1) 2
    2 2 (- 2) (- 2) (- 2) 3 (- 3)])
(type NoteState
   (struct
      noteIndex i32
      octave i32
      note i32
      dur f64))

(def nextNote (state NoteState -> NoteState)
   (var disp i32 (at (cast i32 (% (prng) (len dispTable))) dispTable))
   (set (field noteIndex state) (+ (field noteIndex state) disp))
   (while (< (field noteIndex state) 0)
      (set (field noteIndex state) (+ (field noteIndex state) (len scale)))
      (set (field octave state) (- (field octave state) 1)))
   (while (>= (field noteIndex state) (len scale))
      (set (field noteIndex state) (- (field noteIndex state) (len scale)))
      (set (field octave state) (+ (field octave state) 1)))
   (if (<= (field octave state) (- 2)) (set (field octave state) (- 1)))
   (if (>= (field octave state) 2) (set (field octave state) 1))
   (set (field note state) (+ (at (field noteIndex state) scale) (* (field octave state) 12)))
   (set (field dur state) (/ (cast f64 (+ 1 (% (prng) 2))) 4))
   (return state))

/* play note helper */
(def playNote (voice (slice u8) state NoteState transpose i32 scale f64)
   /* send note on */
   (var note i32 (+ (field note state) transpose))
   (var packet (slice u8) (osc:msg_format backing voice note 127))
   (net:socket_send socket (ref addr) packet)
   /* pause for the duration of the note, then send note off*/
   (pause (- (* scale (field dur state)) 0.05))
   (set packet (osc:msg_format backing voice note 0))
   (net:socket_send socket (ref addr) packet)
   (pause 0.05))
```

Figure 5.1: Quadrant demo program - helpers procedures

```
(var backing (slice u8))
(var socket net:Socket)
(var addr net:Address)
/* entry point */
(def start ()
   /* open a socket to talk to Max/MSP */
   (set socket (net:socket_open 0))
   (set (field ip addr) 2130706433)
   (set (field port addr) 8000)
   (set backing (mem:alloc_slice 1024 u8))
   /*lead voice */
   (var state NoteState)
   (var t (future void)
      /* main flow */
      (flow @(tempo #tempo_curve)
         (var t1 (future void)
            /* lead voice */
            (flow
               (var state NoteState)
               (for (var i i32 0)
                    (< i 30)
                    (set i (+ i 1))
                  /* get next note and play it */
                  (set state (nextNote state))
                  (playNote "/osc/lead" state 0 1))))
         (var t2 (future void)
            /* bass voice */
            (flow
               (var state NoteState)
               (for (var i i32 0)
                    (< i 15)
                    (set i (+ i 1))
                  /* get next note and play it */
                  (set state (nextNote state))
                  (playNote "/osc/bass" state (- 12) 2))))
         /* wait for both voices */
         (wait t1)
         (wait t2)))
   (wait t)
   /*release resources */
   (net:socket_close socket)
   (mem:free_slice backing))
```

Figure 5.2: Quadrant demo program - entry point

# Chapter 6

# Compiler and Runtime Implementation

Temporal scenarios written in the Quadrant editor are compiled to a bytecode representation, and run in a virtual machine. The execution generates feedback data that is used to monitor the progression of the scenario inside the editor.

We describe the compiler pipeline in section 6.1. We then cover the implementation of the virtual machine in section 6.2. Finally, in section 6.3, we explain the execution tracking mechanisms, and how this information is used to implement various live monitoring indicators.

## 6.1 Compiler Pipeline

Modules are processed by several pipelined stages to produce a program that can be executed by the virtual machine (Figure 6.1).

- The parsing stage syntactically validates cell trees and produce an untyped abstract syntax tree.
- The checking stage checks types and symbols, producing a typed intermediate representation along with symbol tables.
- The generating stage lays out types and variables, and assembles modules into an bytecode image that can be loaded by the virtual machine.

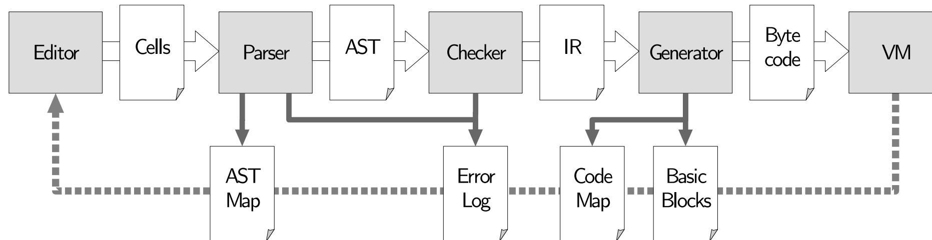



Figure 6.1: Quadrant Pipeline

### 6.1.1 Modules Handling

Quadrant programs are made of modules, which currently correspond to one cell tree. From an implementation point of view, each module is a bundle of resources that include code representations at each stage of the pipeline (i.e. a module "contains" the cell tree, as well as the abstract syntax tree and the intermediate representation associated with it).

The *workspace* is a container that holds modules currently loaded in the editor, as well as modules loaded as a result of the execution of an import directive during a prior compilation cycle. Modules can be backed by files or be transient modules, only existing in the workspace.

The user can select a "main" module, which is the module where checking starts, recursively loading all imported modules into the workspace. The workspace maintains a build list, which is a list of all modules used in the current program. When a module is edited, it is re-parsed in order to provide up-to-date syntactic information to the editor (e.g. for auto-layout or syntax highlighting). If that module is in the build list, the program is then re-checked, and if there is no errors, a new image is generated.

### 6.1.2 Parser

The first stage consists of a hand-written recursive descent parser operating on cell trees, defined in `parser.cpp`[1]. The parser traverses cells hierarchies, checks the syntactical soundness of the forms defined by the language and produces an abstract syntax tree.

The parser is "forgiving", in the sense that it will generate placeholders or error nodes for constructs that are absent or faulty, in a form that still allows

[1]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/parser.cpp

further processing of the code. The error nodes also record what construction was expected at this location, in order for the editor to generate placeholder cells and auto-completion suggestions.

The parser also updates each cell it traverses to point to its resulting AST node, if any. This association is used by the editor to query syntactic properties of cells, in order to perform auto-layout and syntax highlighting.

### 6.1.3 Checker

The checker is defined in `checker.cpp`[2]. It traverses and type checks the abstract syntax tree to create an intermediate representation of the module's code, as well as symbol scopes. A symbol scope is a hash maps of symbols associated to a lexical scope of the program. A symbol refers to an entity such as a variable, a procedure, a type definition, or an imported module.

#### Checking Order

Due to module imports, the parsing and checking stages are actually intertwined. A module is processed in the following order:

- The module is first parsed.
- For each import directive in the AST, the associated modules are loaded and processed.
- The rest of the AST is then checked.

The compiler starts this recursive process at the main module, which is a module selected by the user as the "root" of their program.

Checking inside each module happens in that order:

- The checker first checks type definitions and registers symbols for them in the global scope.
- It then checks signatures of foreign procedure declarations and registers procedure symbols for them in the global scope.
- Next, global variables are checked and registered in the global scope.
- Finally procedures are checked:

[2]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/checker.cpp

- Procedure signatures are checked first, and procedure symbols are created.
- Bodies of non polymorphic procedures are checked and attached to the procedure's symbol.

This process is driven by the `parse_and_check_recursive()` procedure found in `workspace.cpp`[3].

**Polymorphic Procedures Instantiation**

As well as checking code, the checker is responsible for generating concrete procedure instances from polymorphic procedures and arguments concrete types. Polymorphic procedures' symbols are created normally as part of checking all procedure's definitions, but only the signature is checked. The procedure symbol is associated with the body's untyped abstract syntax tree.

When a call to a polymorphic procedure is encountered, the checker calls a pattern-matching function to map each polymorphic identifier used in the procedure's signature to a concrete type. The checker then looks for an existing instance of the polymorphic procedure with the same list of bindings. If it is not found, a new instance symbol is created and the body is checked using the bindings' concrete types in place of the polymorphic identifiers. The call is then bound to the found or newly instantiated procedure symbol.

**Checking Flow Blocks**

Flow blocks create an anonymous procedure symbol. The return type of the block is inferred and wrapped in a `future` type. The block is checked as the procedure's body, and the invocation site of the flow block in its parent procedure is replace by a task call instruction referencing the anonymous procedure.

If the flow block uses a local symbol from its parent procedure, that procedure is marked as captured. A capture pointer symbol is created in the the flow's scope, pointing to the captured symbol (which can itself be a capture pointer, etc.). Eventually the chain of capture pointers leads to the captured variable. All capture pointer variables are kept in a list to allow the generator to output code to store the address of the original variable in the capture reference variable when entering the task's procedure.

[3]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/workspace.cpp

#### Checking Tempo Curve Editors

The checker automatically registers a few built-in types representing the layout of the curve structure in memory. In order to be able to modify the memory representation of a tempo curve while the program is running[4], the checker actually creates two symbols when checking a curve editor:

- An anonymous global variable is created to hold a pointer to the curve data.
- A widget symbol is created to refer to both the curve editor and the global pointer.

These two symbols are used later by the generator and the runtime to allow updating the curve data while running (see subsubsection 6.1.4 and subsection 6.3.4). The checker then creates an intermediate representation node for the anonymous curve pointer and returns it to the parent expression.

#### Errors Logs

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

### 6.1.4 Generator

When the program has been successfully parsed and checked, the generator, defined in `generator.cpp`[5], uses the intermediate representation to generate a serialized representation which can be loaded and executed by the virtual machine. The program image produced by the code generator contains several sections:

- `foreign table`: this section contains a list of foreign libraries to load and symbols to import from those libraries.

[4]Currently the time transformation specified by a tempo curve is applied when the associated task starts. Allowing the tempo curve to change while the task is running is possible, but creates a discrepancy between the actual (time, position) coordinates of the task and those same coordinates according to the tempo curve. This discrepancy could be resolved by silently offsetting either the tempo curve or the current task's position so that those coordinates match, as we do for catch-up curves.

[5]https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/generator.cpp

- `rodata`: this section contains static data such as string literals or runtime type informations.
- `bss`: this 8 bytes integer stores the size of the `bss` section, which is a zero-initialized section of the runtime program's memory.
- `widgets`: this section contains static data for editor widgets such as tempo curves.
- `code`: this section contains the bytecode of the program's procedures.
- `modules`: this section contains module names.
- `locations`: this section contains location entries, which associate bytecode offsets to module indices and cells identifiers.
- `target blocks`: this section contains block entries, which associate target offsets to a start and end code location. These are used by the editor to highlight portions of the source when they are executed by the virtual machine

The generator first first generates static data and runtime type information for types that are involved in boxing operations. Finally, it generates procedures bytecode, allocating local variables addresses inside the procedure's stack frame according to their size and alignment.

### Types Memory Layout

Types memory layout are actually computed beforehand during checking. The generator then lays out variables in their respective stack frames using the types' sizes and alignments. Memory layout follows a set of alignment and padding rules:

- Primitive types are aligned on their size.
- Compound types are aligned on the largest alignment of all their members.
- Compound types members are laid out in the order of declaration, starting at offset 0.
- Compound types are padded to fit in the smallest possible multiple of their alignment.

These match the alignment rules of common C compilers[6], which makes it easier to pass objects to and from foreign code.

### Generating Flow Blocks

Flow blocks are generated as a call to an anonymous procedure, but use a `task_call` instruction instead of a `call` instruction. The flow's anonymous procedure itself is generated as a normal procedure, except for a preamble that is in charge of setting up pointers to the variables captured by the flow.

For each captured variable, the generator pushes the address of the capture pointer, and emits a `capture` instruction with two immediates, namely the number of frames to climb back in the call stack to find the captured variables, and the offset of the captured variable in its stack frame. It then emit a `store` instruction to store the address of the captured variable in the local capture pointer.

Regular procedures whose stack frame can be captured by a flow block are also called using the `task_call` instruction, and are treated much like flows from the runtime's point of view. This is to allow these procedures to outlive their caller while their stack frame is being capture by a child flow.

### Generating Tempo Curves

When a flow block has a tempo attribute, the generator first pushes the `task` instruction, then pushes a `dup` instruction to duplicate the task handle produced by the `task` instruction. The generator then pushes the address of the tempo curve's global pointer, followed by a `tempo` instruction. This will instruct the virtual machine to apply the tempo curve to the newly created task. The curve data itself is serialized and added to the `widgets` section of the program image.

[6] Although the C specification doesn't mandate these rules as such, and compiler attributes can modify these rules, they naturally arise from the constraints of aligning memory accesses and minimizing structures size.

## 6.2 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[7]. The bytecode format is defined in `bytecode.h`[8] and the virtual machine's operations are defined in `vm.cpp`[9].

### 6.2.1 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, 2, 4, or 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.

### 6.2.2 Program Loading

Upon launching the virtual machine, the following steps are performed:

- The loader traverses the foreign table and loads the required libraries. For each library, it imports procedures symbols and prepares the foreign function interfaces needed to call them.
- The loader copies the `rodata` section into VM's memory and allocate and zero-initializes the `bss` section.
- The loader allocates memory in the VM heap for widgets structures, copies static widgets data, and sets widgets global pointers to point to their respective heap-allocated representations. It also builds a widget map linking the module index and cell identifier of each widget to the address of its global pointer.
- The editor builds a map of source locations and target blocks keyed by bytecode offsets.

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

[8] https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/bytecode.h

[9] https://forge-2.ircam.fr/fouilleul/thesis_quadrant/-/blob/master/src/vm.cpp

### 6.2.3 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 (Figure 6.2):

- `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. In the current implementation we use `dlmalloc` (Lee, 1996), which allows allocating from a user-specified memory block (using its `mspace` API).

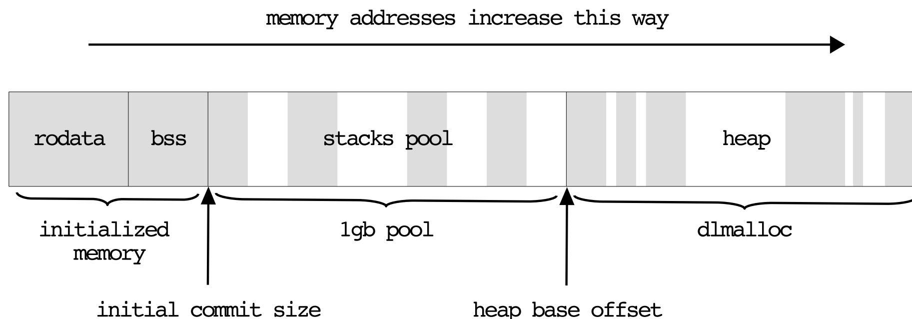



Figure 6.2: Quadrant Address Space Layout

### 6.2.4 Task Structure

The runtime counterpart of a flow (a language-level concept) is an entity called a *task*. Each task is executed by the virtual machine in its own scheduler fiber (see section 4.5). When a task execution reaches a suspending instruction such as a `pause` or `wait`, its fiber yields to the scheduler, which then picks the next fiber (and thus the next task) to resume.

The runtime 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 runtime 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 runtime 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 other active tasks capturing 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 runtime can recycle the task structure.

### 6.2.5 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.
- `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 6.3). 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.

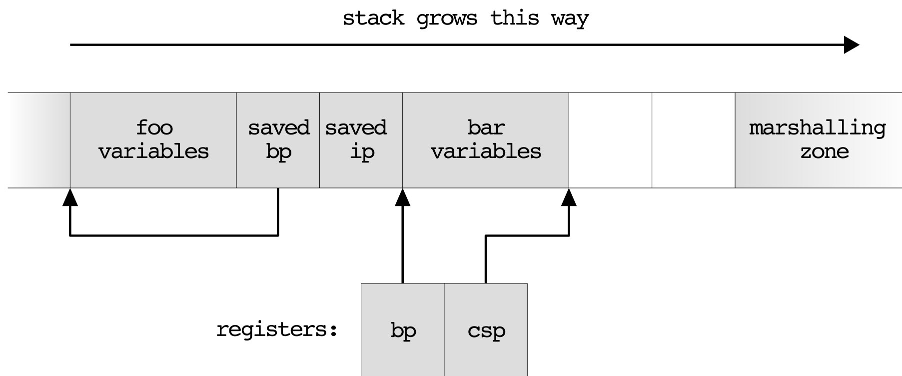



Figure 6.3: Control Stack Layout

### 6.2.6 Calling Convention

#### 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. The `call` opcode itself 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.

#### 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 then evaluated and collected to the marshalling zone the same way as a normal call. The `task` opcode then creates a child task. It allocates

a new operand and control stacks for that 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 virtual machine also creates a new native fiber to run the task, and allocates a native stack for that fiber. The caller then yields to the scheduler to pass execution to the new fiber.

In case the task had a tempo attribute, a `dup` instruction will first duplicate the new task's future at the top of the operands stack. The curve's global pointer address is then pushed to the operand stack. The `tempo` instruction then dereferences the pointer to get the curve data address, and applies the tempo curve to the task.

#### 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.

#### Foreign Calls

The generator associates an integer index to each foreign procedures, 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`[10]) into an array.

A foreign call starts 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.

## 6.3 Execution Monitoring and Control

Execution monitoring is achieved by running the virtual machine and the editor in separate threads and have them communicate via message passing

[10]https://github.com/libffi/libffi

using a pair of wait-free ring buffers. The `control` buffer is used to pass commands from the editor to the virtual machine, and the `feedback` buffer is used to pass feedback informations from the virtual machine to the editor.

### 6.3.1 Execution Blocks

An execution block 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 loading the program, the editor processes this table to produce a map for fast retrieval of cell ranges from bytecode offsets.

When executing a jump or a yielding instruction, the virtual machine sends a `trace` message to the editor, containing the target offset of the jump. The editor then uses the blocks map lookup the corresponding range of executed cells, and displays a flashing 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 virtual machine to emit a `trace` message.

### 6.3.2 Progress Reports

The scheduler runs a special fiber called `vm_progress_report_task()` at a fixed frequency to monitor the progress of runtime tasks.

For each such task, the monitoring fiber collects a call stack by walking back the task's control stack and recording the saved instruction pointer for each stack frame (which corresponds to the the bytecode offset immediately following the call-site for that frame). The monitoring fiber forms `progress` message containing the status of the task, the time remaining, and the call stack, and sends it to the editor. The editor then uses the code map to display progress wheels, spinning wheels or standby icons next to all call sites along the call stack of a suspended task.

### 6.3.3 Standby Triggers

When the cursor is inside or on the left of 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 the `standby` form to the virtual machine. The virtual machine then resumes every task that is on standby at that particular bytecode location. This provides a way to manually pace the progress of the scenario from within the editor.

### 6.3.4 Updating Tempo Curves

When a tempo curve is modified in a curve editor while the scenario is running, the editor sends an `update_tempo` message containing the serialized curve data, along with the module index and cell index of the editor, to the runtime. Upon receiving this message, the runtime uses the widget map to get the global pointer of the curve. It then frees the memory allocated to the old curve, and allocates a new block of memory to hold the new curve data. The runtime curve representation is deserialized into this new block. Finally, the curve's global pointer is updated to point to the new block.

## 6.4 Conclusion

We described the architecture of the compiler pipeline and some aspects of its implementation. We then explained how the runtime executes compiled programs. We finally detailed the feedback mechanisms which allow live monitoring inside the editor.

Currently, the whole program is re-parsed and re-checked when a modification is done (with the exception of curve editors). This isn't a problem for moderately sized programs, because the absence of a lexing pass and the simplicity of the language grammar and type system allow sufficiently fast compilation times. However, the structured nature of program representation could enable fine grained incremental compilation (for instance, directly at the form level). This would allow larger scenarios to be edited with the same immediate semantics-aware feedback as smaller ones.

Incremental compilation would also open the way to hot patching running scenarios. Hot patching could be done easily at the procedure level, by appending the byte code for the new version of a procedure at the end of the byte code section, and overwriting the start of the old version with a jump to the new version. This is sufficient for procedures that execute atomically, but

can lead to unexpected behaviour for long-lived tasks that get suspended and resumed, since the new version is executed only the next time the procedure is entered.

Patching smaller grained areas, such as execution blocks, may be tractable in the same way but raise issues regarding the consistency of the procedure's stack frame layout. For instance, inserting a variable declaration at the beginning of the procedure should impact all variable address computations in subsequent execution blocks. A dependency analysis could determine which blocks need to be re-generated, but the old stack frame would also need to be marshalled into the new layout. This would require an understanding of high-level data semantics to be baked in the runtime: for instance, the patching mechanism would need to know if a given variable holds a pointer to another variable in the frame, in order to patch it. Variable captures only complicate this picture.

Hot patching at either procedure or basic execution block level, or even at the module level, raise the same issue regarding the consistency of global data addressing. Additionally, it is unclear how a change in a data type definition should be handled with regards to existing variables of this type (e.g. should added fields be initialized to a default value? should renamed fields loose or preserve their value? etc.).

We didn't tackled these challenges in this work, but hot patching would obviously be a keystone in allowing a really tight feedback loop during creation and rehearsal of temporal scenarios. It would also open up the environment to live coding use cases. However, we must acknowledge that combining the conflicting requirements of live coding with those of a statically typed language compiled to a low-level virtual machine, may require a reassessment of the design decisions underpinning Quadrant's architecture. This is of course an important avenue for future work.

# Chapter 7

# Surrounding Infrastructure

In order to produce actions impacting the show, such as dimming a light or playing a sound, Quadrant must be able to connect and communicate with various devices and software. In this chapter we discuss our work on some components of a distributed temporal interaction infrastructure. These are not all part of Quadrant *per se*, but are meant to form the basis of a surrounding ecosystem aimed at easing the orchestration of the multiple software and technical artifacts used in live performances and installations.

We first describe the core modules currently available in Quadrant, and detail the workings of some of them (section 7.1). We then detail the external services composing our early interaction ecosystem. These include a service discovery agent (section 7.2), a leader election protocol (section 7.3), and a clock synchronization service (section 7.4). Finally, we present an artistic application of these components, that was conducted to inform and drive their development (section 7.5).

## 7.1 Core Modules

Quadrant currently comes with several "core" modules, that can be seen as the embryo of a standard library.

- The `mem` module implements memory allocators.
- The `strings` module provides helpers for manipulating strings.
- The `fmt` module provides a formatted logging API.

- The `net` module gives access to network sockets.
- The `osc` module allows composing and parsing OSC messages.
- The `sync` module wraps the internal phase synchronization API of Quadrant's scheduler.

We detail some of these modules below.

### 7.1.1 Memory

The `mem` module provides memory allocators. It imports two foreign procedures from the runtime, `qvm_alloc()` and `qvm_free()`, that respectively allocate and free memory from the virtual machine heap, using `dlmalloc`.

On top of this internal API, the module provides typed allocation helpers for single objects and slice allocations, taking advantage of Quadrant's explicit polymorphic procedures (Listing 7.1).

```
(def alloc (size u64 -> (ptr u8)))
(def free (ptr u8))
(def alloc_obj ($T typeid -> (ptr $T)))
(def free_obj (ptr $T))
(def alloc_slice (count u64 $T typeid -> (slice $T)))
(def free_slice (s (slice $T)))
```

Listing 7.1: Typed object and slice allocators.

The module also provides an API to create and allocate from memory arenas[1] (Listing 7.2).

### 7.1.2 OSC

The `osc` module implements procedures for composing and parsing OSC messages.

#### Data Interchange Discussion

Although we settled on OSC as a first data interchange format to implement, several formats where considered:

[1] An arena, bump allocator, or linear allocator, is a simple and fast allocator that allows pushing heterogeneous objects or arrays one by one, and clearing them all at once.

```
(def arena_create (size u64 -> Arena))
(def arena_destroy (arena Arena))
(def arena_alloc (arena (ptr Arena)
                  size u64
                  -> (ptr u8)))
(def arena_alloc_obj (arena (ptr Arena)
                      $T typeid
                      -> (ptr $T)))
(def arena_alloc_slice (arena (ptr Arena)
                        count u64
                        $T typeid
                        -> (slice $T)))
(def arena_clear (arena Arena))
```

Listing 7.2: Arena allocator.

- Textual formats such as XML (Bray et al., 2008) or JSON (Bray, 2014) have the advantage of being human-readable with a simple text editor. They however require more complex parsers and are generally less performant than binary formats.
- Interface definition languages such as OMG IDL (Open Management Group, 2018) or Google Protocol Buffers (Google, 2008) allow more performant implementations, and automatically generate serialization and deserialization code. However they require interface description files to be kept in sync between modules, and changing these specifications include a build step. In the context of a creative process where these specifications can change very often, this can be impractical.
- Domain specific protocol such as MIDI (MIDI Manufacturers Association, 1982) or OSC (Wright, 2002) are often used in musical applications. The MIDI protocol seems to specific to instrument control to be used as a general-purpose data interchange format. OSC affords a simple and somewhat extensible solution. It has the advantage of being very widespread in audio application and musical controllers.

OSC does have some drawbacks, especially when it comes to exchanging structured data. In fact it was designed to control musical application by sending commands along with typed parameters, and thus models function calls rather than structured data. This manifests in several ways:

- The only way of expressing hierarchical data is through the notion of *bundle*, which is a collection of messages. This could allow expressing key-value structures (such as C structures of Python dictionaries), by having each message in a bundle being keyed by its address pattern. However, this doesn't allow nesting key-value structures, since a bundle itself has no address pattern[2]. Thus constructing complex structures requires additional wrapping messages, which quickly become tedious.
- OSC lacks a proper notion of arrays, and only has begin and end *markers* that can delineate collections of heterogeneous objects. In particular, this does not afford random access to an element of an array.

Additionally, despite its claims of flexibility, the OSC design is not actually amenable to clean extension and ascending compatibility. For example, an OSC message doesn't embed the size of the arguments it contains, and the protocol relies on the tag to implicitly encode this information. As such, an argument of unknown type can't be skipped. In practice most implementations simply bail out when they encounter unknown type tags. The protocol is simultaneously underspecified in a number of ways, for example as it pertains to temporality, the semantics of bundles, and the semantics of its `timetag` type (Freed & Schmeder, 2009).

Finally, the encoding of OSC messages makes composing messages by adding successive arguments inefficient, since the location of arguments in the serialized buffer isn't known before the whole message is composed. The use of zero-terminated, 4 bytes-aligned strings and the choice of big-endian encoding also contribute to make it a sub-optimal data interchange format.

Despite its flaw, it relative ease of use for simple scenarios, and its ubiquitous adoption in music softwares makes it a good first protocol to add to the Quadrant core library.

[2]Libo (McCallum, 2015) is an implementation of OSC that extends the protocol with a bundle type for messages arguments. It is used in Odot (Maccallum et al., 2015), a toolkit that heavily relies on such nesting to build a computing model on top of OSC structures. However, we think that embedding bundles as message arguments is not exactly the same as having simple key-value structures, since the arguments themselves have no names, and thus multiple bundle arguments can only be keyed by the name of their parent message.

#### Implementation And Performance

We evaluated several existing libraries, namely `libo` (McCallum, 2015), `liblo` (Harris & Sinclair, 2004), `oscpack` (Bencina, 2013) and `rtosc` (McCurry, 2018). They exhibit widely different feature sets, but to various degrees, they all conflate notions related to the data format (parsing and composing messages), the notions related to the transport protocol (sending and receiving messages), and those related to the semantics (dispatching messages and matching them to function calls). To the exception of `oscpack`, their design and/or API choices also has performance costs.

Since we only needed a lightweight and fast way of marshalling OSC messages from and to byte buffers, we ultimately decided to write our own internal C library, nicknamed `blitz`. The `osc` module leverages that library through the foreign system.

Our implementation composes and parses messages and bundles in-place, using memory buffers provided by client code. Structures exposed by the API only contain pointers to particular slots into these buffers and do not "own" memory. This way the library completely avoid memory allocations, which was instrumental in achieving good performance.

The library affords two APIs for building OSC elements (i.e. messages or bundles). The `push` API (Listing 7.3) allows composing elements step-by-step, by successively pushing typed arguments. The `format` API (Listing 7.4) allows creating OSC elements in one go, using a format string and variable arguments, similar to `printf`. This is useful when all arguments are immediately

```
OscErrCode OscBeginMessage(osc_element* elt,
                           const char* addressPattern);

OscErrCode OscEndMessage(osc_element* elt);
OscErrCode OscPushInt32(osc_element* elt, int32 i);
OscErrCode OscPushFloat(osc_element* elt, float f);
OscErrCode OscPushString(osc_element* elt, const char* s);
//...
```

Listing 7.3: Blitz Push API (excerpt).

Likewise, parsing can be done in two different ways. The `iterator` API (Listing 7.5) uses an iterator type which allows traversing the structure of an OSC packet and extracting address patterns and arguments. The `scan`

```
OscErrCode OscFormat(osc_element* elt,
                     uint32 size,
                     char* buffer,
                     const char* pattern,
                     const char* typeTags,
                     ...);
//...
```

Listing 7.4: Blitz Format API (excerpt).

API (Listing 7.6) allow checking and extracting arguments of a message in one go using a format string, similar to `scanf`.

```
OscErrCode OscParseMessage(osc_msg* msg,
                           int32 size,
                           const char* packet);

osc_arg_iterator OscArgumentsBegin(osc_msg* msg);
osc_arg_iterator* OscArgumentNext(osc_arg_iterator* arg);
OscErrCode OscAsInt32(osc_arg_iterator* arg, int32* i);
OscErrCode OscAsFloat(osc_arg_iterator* arg, float* f);
OscErrCode OscAsString(osc_arg_iterator* arg, const char** s);
//...
```

Listing 7.5: Blitz Iterator API (excerpt).

```
OscErrCode OscScan(osc_msg* elt,
                   const char* address,
                   const char* typeTags,
                   ...);
//...
```

Listing 7.6: Blitz Scan API (excerpt).

We carried out a benchmark comprising our implementation and the aforementioned OSC libraries. We measured the following tasks:

- Composing an OSC message.
- Composing an OSC bundle containing two messages.

- Parsing an OSC message and extracting its address pattern, type tags and arguments.
- Parsing an OSC bundle containing two messages, and extracting their address pattern, type tags and arguments.

The message we used had the address pattern `/foo/bar`, the arguments `"Hello, world !"`, followed by a 64 bits integer and a 64 bits floating point number. Each library was compiled from sources with `-O3` optimization level, and statically linked to the test program. Tasks were repeated a hundred million times in a loop and the total execution time was recorded.

Table 7.1 shows the benchmark results. For each task, we give the total execution time, the number of task iteration per seconds, the mean duration of one iteration, and the speed-up ratio compared to the slowest implementation. We also mention the speed-up factor of our implementation compared to the fastest alternative (which was `oscpack` in all cases). `liblo` doesn't have a way of directly parsing bundles, so it doesn't appear for this particular task.

It is worth remembering that `libo` and `liblo` have a widely more versatile feature set, allowing to arbitrarily insert and deletes elements from messages. In the case of `libo`, the library is meant to support a full expression language based on OSC bundles (Maccallum et al., 2015). These wide and generic feature sets are paid in performance cost, which might be a reasonable tradeoff for their intended use-case.

## 7.2 Service Discovery

Service discovery allows locating available resources on a network, such as shared files or printers, without needing manual configuration. The basic idea behind service discovery is that processes or devices providing a service can register themselves under a known name, and that processes needing a given service can query it by name and somehow map that name to its physical location.

Several dedicated protocols have been designed for service discovery, such as SSDP (Albright et al., 1999), WSSD (OASIS, 2009), or NetBios (NetBIOS Working Group, 1987). Another widespread strategy is to use the mDNS records of a domain name server (Cheshire & Krochmal, 2013). For example, DNS-based service discovery was originally implemented by the Bonjour

| Implementation | time (s) | Op length ($\mu$s) | Throughput (op/s) | Speedup vs. slowest | Speedup vs. oscpack |
|---|---|---|---|---|---|
| composing $1 \times 10^8$ messages | | | | | |
| Blitz push | 4.38 | 0.044 | 22823059 | 30.11 | 1.52 |
| Blitz format | 4.14 | 0.041 | 24154397 | 31.86 | 1.61 |
| oscpack | 6.68 | 0.067 | 14980604 | 19.76 | 1.00 |
| rtosc | 7.97 | 0.080 | 12549376 | 16.55 | 0.84 |
| liblo | 75.68 | 0.757 | 1321303 | 1.74 | 0.09 |
| libo | 131.91 | 1.319 | 758065 | 1.00 | 0.07 |
| Composing $1 \times 10^8$ bundles | | | | | |
| Blitz push | 10.26 | 0.103 | 9745841 | 24.92 | 1.40 |
| Blitz format | 9.89 | 0.099 | 10107700 | 25.85 | 1.45 |
| oscpack | 14.36 | 0.144 | 6964290 | 17.81 | 1.00 |
| rtosc | 34.58 | 0.346 | 2892121 | 7.40 | 0.42 |
| liblo | 235.76 | 2.358 | 424164 | 1.08 | 0.06 |
| libo | 255.74 | 2.557 | 391028 | 1.00 | 0.06 |
| Parsing $1 \times 10^8$ messages | | | | | |
| Blitz iterator | 2.94 | 0.029 | 34013826 | 20.49 | 1.44 |
| Blitz scan | 3.14 | 0.031 | 31875865 | 19.20 | 1.35 |
| oscpack | 4.22 | 0.042 | 23689203 | 14.27 | 1.00 |
| rtosc | 13.18 | 0.132 | 7584987 | 4.57 | 0.32 |
| liblo | 60.24 | 0.602 | 1660161 | 1.00 | 0.07 |
| libo | 30.69 | 0.307 | 3258812 | 1.96 | 0.14 |
| Parsing $1 \times 10^8$ bundles | | | | | |
| Blitz iterator | 7.74 | 0.077 | 12918467 | 6.44 | 1.35 |
| Blitz scan | 8.52 | 0.085 | 11732967 | 5.85 | 1.23 |
| oscpack | 10.44 | 0.104 | 9574139 | 4.77 | 1.00 |
| rtosc | 34.76 | 0.348 | 2876787 | 1.43 | 0.30 |
| libo | 49.85 | 0.499 | 2005921 | 1.00 | 0.21 |

Table 7.1: Comparison of performances of several OSC libraries.

protocol (Apple, 2003) in Apple products and has been adopted by a number of other vendors.

While adopting an existing protocol such as Bonjour seems like an obvious choice, it necessitates a pre-existing infrastructure, namely a DNS server. We would prefer a system that doesn't force users to setup and configure such a server for small installations. Furthermore, using mDNS records imposes the use of unique service names, whereas relaxing this constraints allows us to very easily implement publication/subscription message passing schemes. We hence opted for an *ad hoc* service discovery system, without precluding the future addition of a Bonjour implementation as an extension module.

### 7.2.1 Discovery, publication, and revocation

A services is identified across the network by a service descriptor, which consists of a name, a type, an globally unique identifier, and an address at which this service can be contacted. A process can use the service discovery API to spawn a "service discovery agent", which is run in a separate thread. It can then publish or request service descriptors through its agent, and can also request to be notified by the agent when a particular service appears or disappears from the network.

The exchange of messages between agents proceeds as follow:

- After being launched, it joins a multicast group to send and receive requests, and opens an ephemeral UDP port to receive responses from other agents. It then sends a series of discovery requests on the multicast address. These requests consist of an OSC message with address pattern `/hello`, and the agent's response port as an argument.
- When an agent receives a `/hello` message, it sends back a series of responses, one for each locally registered service. These responses consist of an OSC message with address pattern `/publish`, whose arguments are the fields of the service descriptor.
- When a process publishes a service to its service discovery agent, the latter sends a series of `/publish` messages for this service on its multicast address.
- When a process stops providing a service, it can revoke its service descriptor from its service discovery agent. The latter then multicasts a series of `/revoke` OSC messages with the service's identifier as argument.

### 7.2.2 Congestion reduction

Discovery service messages are sent as a series of redundant messages, in order to account for a reasonable probability of packet loss or agents' temporary downtime. In order to avoid network congestion peaks, especially when a lot of service discovery agents are spawned all at once, the interval at which these messages are sent follows a geometric series with a random initial delay. Publication and revocation messages are send a limited number of time. Discovery messages are sent according to the geometric series, and then at a regular interval past a given threshold. This ensures discovery agents keep an up to date list of the services available on the network (up to a certain temporal granularity), even if they are temporarily disconnected and miss an entire series of publication or revocation messages.

### 7.2.3 Expiration

In the case where a process is terminated abruptly, it is possible that the services it published previously are not revoked. Processes that query their discovery agent for a service must hence handle the case where they get back a descriptor associated with an unresponsive service. However, it is desirable to limit the number of such inactive service descriptors in the tables of service discovery agents, because detecting and handling inactive services can be wasteful. For this reason, a service discovery agent attaches an expiration date to each descriptor. This expiration date is postponed each time the agent receives a `/publish` message for this service. When the expiration date is reached, the descriptor is silently dropped from the agent's services table.

## 7.3 Leader Election

Many distributed applications need to select a particular process to coordinate the actions of other processes. For example, clock synchronization protocols may need to elect a clock server on which other processes synchronize. Manually selecting this leader clock server is an additional burden on users. Moreover, statically assigning a leader isn't a robust solutions in the face of network hiccups, or when the composition of the network must be allowed to frequently change. This is why we describe in what follows a leader election protocol, which allows processes to automatically reach a consensus on the selection of a leader process. We latter use that protocol to designate a leader clock in our clock synchronization system.

Our election protocol must account for lost or delayed messages, as well as fallible processes that can abruptly join or leave the election quorum. We build our protocol on the ideas exposed in (Kim & Belford, 1996). However in our use case, we assume that after the election, processes depend only on knowing the elected master, and do not need to know about the active or inactive status of other peers. We thus don't need to broadcast an activity table to all peers. Another difference is that we leverage the discovery service module described in section 7.2 in order to build a list of peers each time an election is started, rather than assuming a pre-established knowledge of all potential peers.

A peer can be in one of the following states:

- `LEADER`: The election round has concluded and the peer has been elected leader.
- `NORMAL`: The election round has concluded and the peer is in agreement with the other peers about the elected leader.
- `CANDIDATE`: The peer has announced its candidacy.
- `COHORT`: The election round is in progress and the peer has chosen a candidate, but it can still amend this choice if it receives a message from a better candidate.
- `CLAIM`: The peer is a candidate and has received enough support (given its knowledge of other peers) to claim victory.
- `CONFIRM`: The peer is confirming its support for its selected candidate after the latter has claimed victory.

When an election is carried through to its end, one and only one peer is in the `LEADER` state and all other peers are in the `NORMAL` state.

Each peer $P_i$ is associated with an address $A_i$ and a unique score $S_i$. Each peer also maintains a variable $B$ storing the tuple $(A_b, S_b)$ designating the current best candidate from its point of view. Each peer also has a timer $T_i$. A peer in the state `CANDIDATE` maintains a list of peers, which indicates for each peer its address, its active or inactive status, and wether this peer accepted the candidate as a leader.

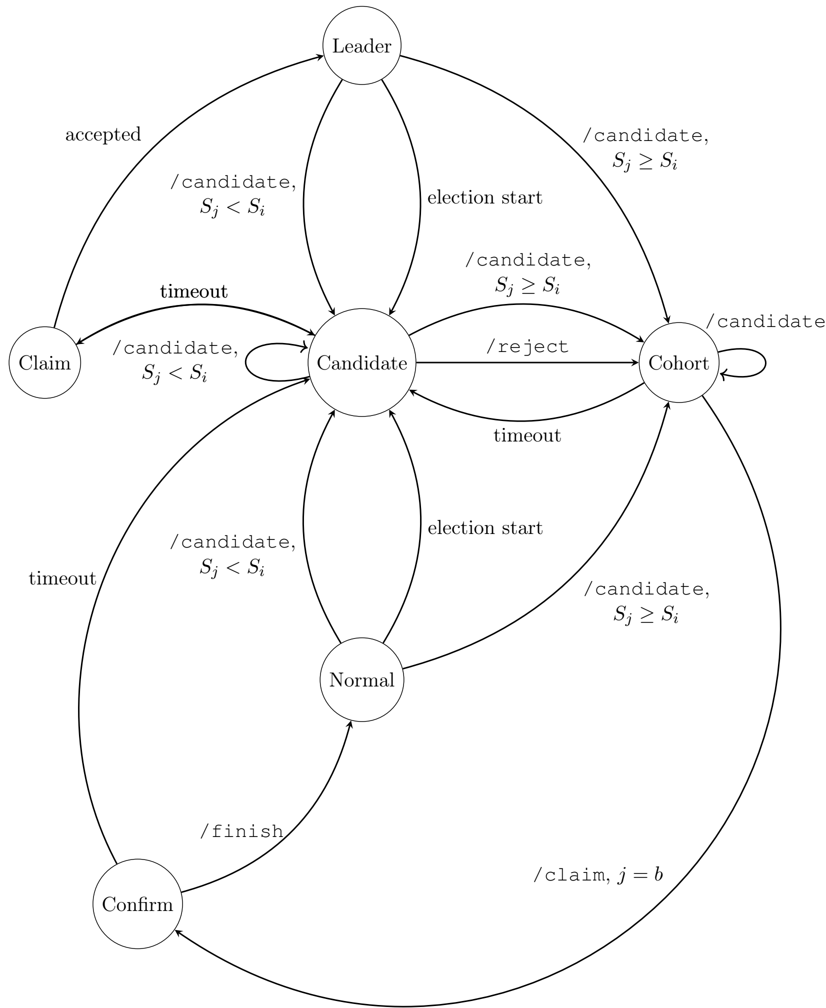



Figure 7.1: Simplified state diagram of the leader election protocol.

Messages exchanged between peers consists of OSC messages whose arguments are tuples of the form $(A_j, S_j)$. For each state, we describe below the possible events or inbound messages and how a peer $P_i$ must react to them. Figure 7.1 shows a state diagram summarizing these transitions. For the

sake of legibility, the diagram has been simplified by omitting some inbound messages that don't trigger a transition, as well as outbound messages sent during each transition.

`NORMAL` or `LEADER`

- Election start, or detected failure of the current leader: $P_i$ asks its service discovery agent a list of potential peers. It sends to each peer a message `/candidate`$(A_i, S_i)$, starts a timer, then switches to state `CANDIDATE`.
- `/candidate`$(A_j, S_j)$: if $S_j < S_i$, $P_i$ starts an election by sending a candidate message as described above. Otherwise it stores $(A_j, S_j)$ as its best candidate, and responds with the message `/ack`$(A_i, S_i)$, starts its timer, and switches to state `COHORT`.

`CANDIDATE`

- `/candidate`$(A_j, S_j)$: if $S_j < S_i$, the candidate $P_i$ replies with a message `/reject`$(A_i, S_i)$. Otherwise it stores $(A_j, S_j)$ in $B_i$ as its best candidate, replies with the message `/ack`$(A_i, S_i)$, starts its timer and switches to state `COHORT`.
- `/reject`$(A_j, S_j)$: the candidate stores $(A_j, S_j)$ in $B_i$ as its best candidate, replies with the message `/ack`$(A_i, S_i)$, starts its timer, and switches to state `COHORT`.
- `/ack`$(A_j, S_j)$: the candidate marks $P_j$ as active in its list of peers.
- $T_i$ timeout: the candidate sends the message `/claim`$(A_i, S_i)$ to all the active peers in its peer list, starts a timer and switches to state `CLAIM`.

`CLAIM`

- `/accept`$(A_j, S_j)$: the candidate $P_i$ marks that peer $P_j$ accepted its candidacy. If all active peers accepted $P_i$ as a leader, $P_i$ sends the message `/finish`$(A_i, S_i)$ to all its active peers, and switches to state `LEADER`.
- $T_i$ timeout: if any of the active peers hasn't accepted $P_i$ as a leader yet, the election is restarted.

`COHORT`

- `/candidate`$(A_j, S_j)$: if $S_j < S_b$, $P_i$ responds with a `/reject`$(A_j, S_j)$ message. Otherwise it stores $(A_j, S_j)$ in $B_i$ as its best candidate, reinitializes its timer, and responds with the message `/ack`$(A_i, S_i)$.
- `/claim`$(A_j, S_j)$: if $j = b$, $P_i$ responds with the message `/accept`$(A_i, S_i)$, starts its timer and switches to state `CONFIRM`. Otherwise, the message is ignored.
- $T_i$ timeout: if $P_i$ hasn't received any `/claim`$(A_b, S_b)$ message when its timer expires, the best candidate is considered failing and the election must be restarted.

`CONFIRM`

- `/finish`$(A_j, S_j)$: if $j = b$, $P_i$ switches to state `NORMAL`. Otherwise, the election is restarted.
- $T_i$ timeout: if $P_i$ has not received a `/finish`$(A_j, S_j)$ message when its timer expires, the best candidate is considered failing and the election must be restarted.

## 7.4 Clock Synchronization

Many messages exchanged between software components of a live artistic performance are time sensitive or carry temporal information. As such, it is particularly important to maintain a consistent notion of time among all participating processes. The distribution of a common clock is not a trivial problem and simple approaches such as (Cristian & Fetzer, 2003; Gusella & Zatti, 1989) don't provide sufficient precision and robustness.

In the field of computer music, this problem is often side-stepped on the grounds that the human ear can't generally distinguish rhythmic displacements or latencies inferior to 10 ms (Friberg & Sundberg, 1993; Jack et al., 2018). We consider this argument moot for several reasons:

- It ignores the fact that small latencies, although not perceived as rhythmic displacements, nevertheless induce phase shifts that audibly manifest as comb filtering, or as spatialization artifacts such as Haas effect (Haas, 1972).
- If care is not taken to compensate them, small instantaneous temporal errors can accumulate over the duration of a show (or even over several

weeks of an art installation exhibition) and result in perceptible timing differences.

- Simple approaches are vulnerable to clock latency and jitter peaks, which can lead to significant errors.
- The psycho-acoustics argument simply isn't relevant in some cases, e.g. control messages that need to be timestamped in order to evaluate them in order.

We think that maintaining a distributed clock with a better precision is an important feature of distributed show-control system. Whereas this seems to have been an under-explored topic in the domain of computer music, it is a well studied problem in networks and distributed systems, and we can benefit from these prior works.

The most widespread standard for clock synchronization in NTP. It has been continuously refined, from its conception (Mills, 1988) until the fourth and current version (Mills et al., 2010), and has proven remarkably robust and precise for a great number of applications. It allows distributing a clock with a few milliseconds precision across the internet, and usually reaches sub-millisecond precision on local area networks (Mills, 2006). An NTP network is an self-organizing hierarchy of client and server clocks. Each node can synchronize as a client on several servers, and redistribute its clock to clients lower in the hierarchy. Clients use several mitigation techniques to correct errors due to the latency, jitter and delay asymmetry of the network.

SNTP (Mills, 1996) is a simplified version of NTP dedicated to peers that have a unique synchronization source and don't redistribute their clock to clients. It only computes an estimation of the phase offset between the server and the client using only the messages exchanged between them, and isn't required to use any kind of mitigation algorithm.

PTP (IEEE, 2008) is dedicated to high precision measures in control systems, and can reach sub-microseconds precisions. It uses a combination of dedicated software and hardware to estimate latency and jitter errors and directly correct the timestamps of messages as they flow across the network.

### 7.4.1 Custom Clock Synchronization Protocol

We built our clock synchronization system on the estimation and mitigation techniques of NTP. However, as in the case of service discovery, using an off-the-shelf implementation isn't very practical, as it requires a pre-existing

infra-structure (separate NTP servers and clients processes) that must be launched and configured manually by users. Furthermore we wanted to be able to tweak some parts of the protocol to make it more suitable to our use case, as discussed in the following paragraphs.

We ignore the aspects of the protocol that pertain to authentication and encryption. We also ignore the stratum hierarchy of NTP, in favour of a simpler scheme where all clients synchronize on a unique clock leader. Since we have a unique leader, we don't need to use the selection, clustering and combining algorithms of NTP, since they deal with producing combined estimates from multiple clock servers. We leverage our service discovery and leader election protocols to designate the clock leader. This makes our custom clock synchronization mechanism tolerant to frequent reconfigurations of the network, and to possible failures of the leader.

For each synchronized process, our protocol maintains *user clock*, derived from the system's clock of the platform on which the process executes. The clock synchronization protocol only adjusts the parameters of this user clock, not the system's clock. Hence, it doesn't require administrative priviledges and doesn't interferes with other processes using the system clock.

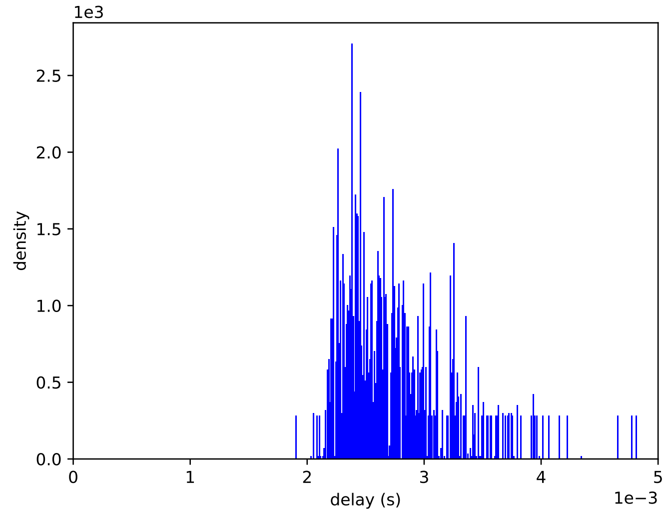
We noticed that some consumer routers go into a low-power mode when no packets are transmitted for some amount of time. The first packet to wake-up the router exhibit a substantially larger delay than subsequent packets. This induces large and inconsistent round-trip time asymmetry, that makes it harder for the algorithms to correctly estimate clock offset and frequency. To alleviate this problem, we modified the on-wire protocol to send *scout messages* before a standard request-response exchange. This wakes-up the routers along the packet's path, before the synchronization exchange takes place.

We don't use the exact same clock filter discipline algorithms as the NTP specification. We instead use a linear regression method, similar to the implementation of the `chrony` time client (Curnow & Lichvar, n.d.). This has the advantage of producing both an offset and a frequency estimate, yields a better asymmetry correction, and simplifies the clock discipline state machine.

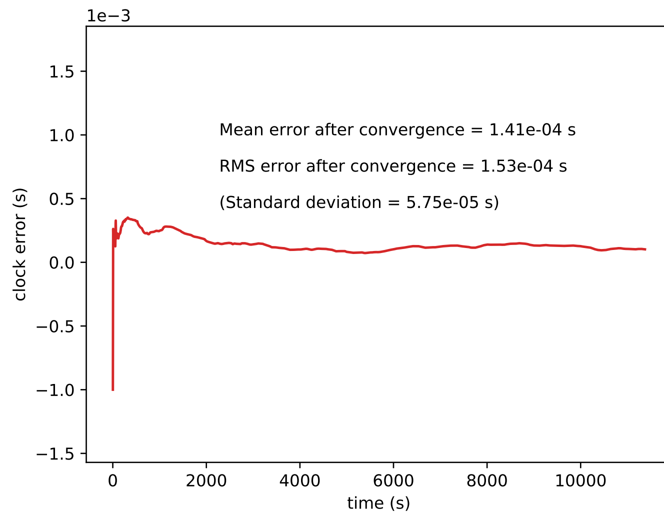
### 7.4.2 Clock Synchronization Simulator

We built a simulator to test and assess the performance of our protocol. The simulator runs offline using virtual clocks and simulates message exchanges

between a clock leader and a client. It uses virtual clocks and inserts virtual delays in the messages transmissions using a pre-recorded trace of delays as measured on a real network. Figure 7.2 and Figure 7.3 show the delay distribution and clock errors, respectively for a trace recorded on a WiFi network, and for a trace recorded on a wired LAN network.

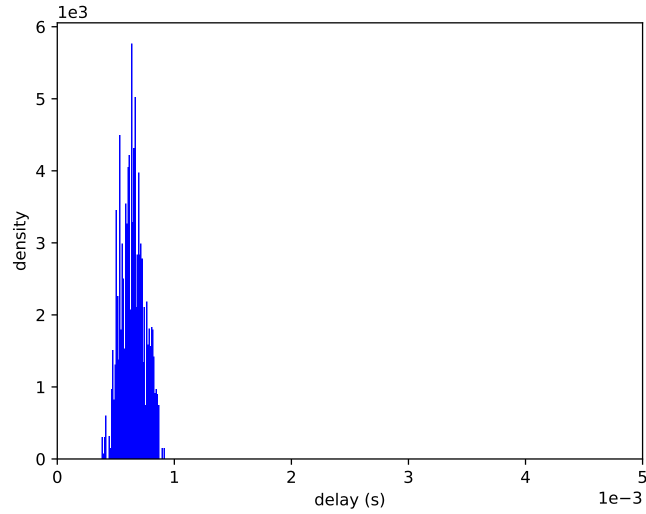



(a) Delay Distribution.

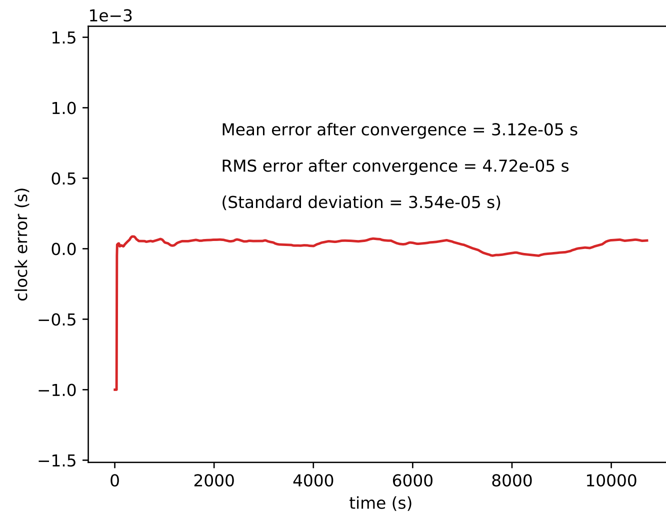



(b) Clock Error.

Figure 7.2: Clock Synchronization Simulation for WiFi Delays.




(a) Delay Distribution.




(b) Clock Error.

Figure 7.3: Clock Synchronization Simulation for Wired LAN Delays.

## 7.5 An Artistic Application

In parallel to Quadrant's development, we bundled the aforementioned protocols as a "distributed interaction" toolbox, in the form of a C library with Python bindings, as well as externals for the Max and PureData audiovisual programming environments. We used the toolbox for a project with composer Pedro Garcia Velasquez, set designer Marion Flament and the ensemble Le Balcon. This project involves a group of robotic arms playing percussion instruments and glass and stone sculptures, along with a chamber ensemble of human musicians (Figure 7.4 and Figure 7.5).

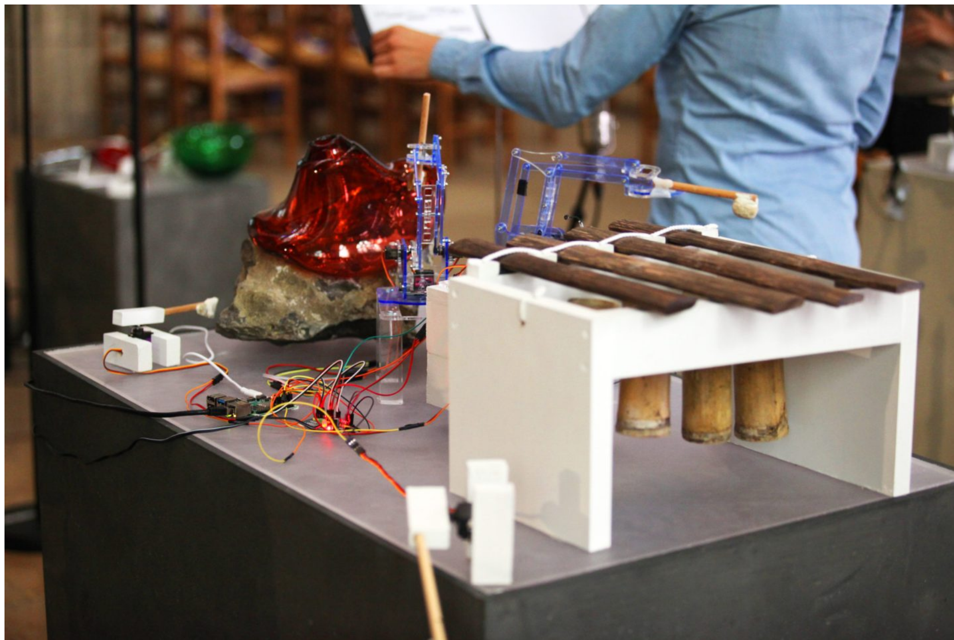


Figure 7.4: Robotic arms used in *La Selva Virgen, en La Selva Oscura*, Marimba de Chonta and Glass Sculptures.

### 7.5.1 Architecture

Robotic arms are moved by servo-motors, which are controlled by Raspberry Pi micro-controllers connected to a WiFi or Ethernet network. Each micro-controller has a Python daemon that handles service discovery, clock synchronization and processes incoming messages. The daemon also loads

Figure 7.5: Robotic arms used in *La Selva Virgen, en La Selva Oscura*, Bowls and Crotales.

a dictionary of gestures, which are timed sequences of servo-motor control signals.

Moves can be triggered by OSC messages sent from a Live sequencer and/or a MIDI keyboard via a MIDI to OSC translation patch. A Max4Live plug-in built around our toolbox externals is responsible for discovering micro-controllers and dispatching OSC messages to the correct robots.

Figure 7.6 shows a diagram of the architecture of the robotic ensemble.

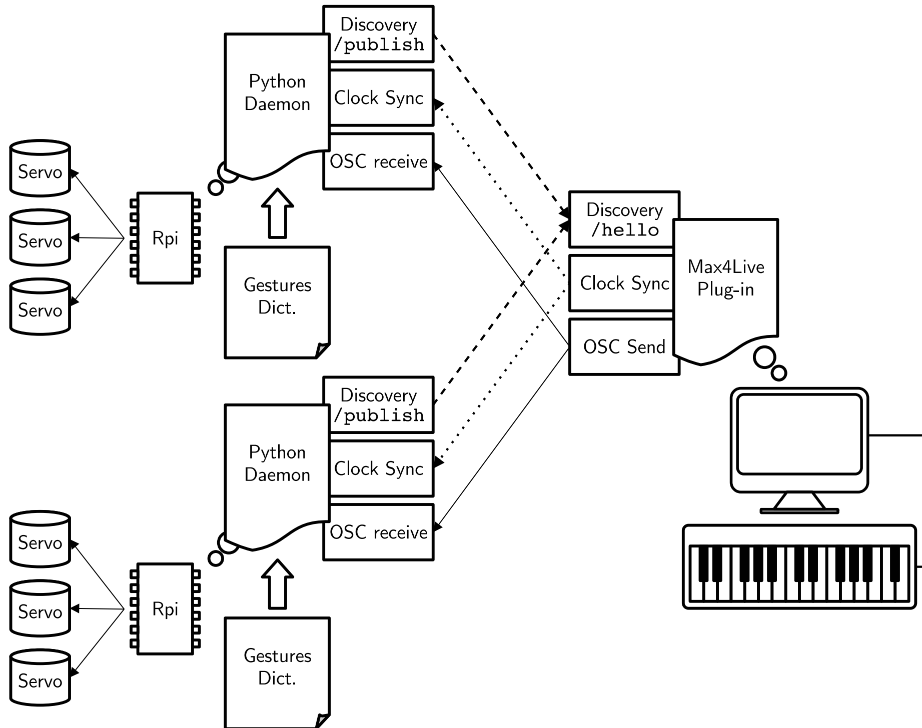



Figure 7.6: Robotic Ensemble Architecture.

### 7.5.2 Gestures

The gesture dictionary is a Python dictionary whose keys are OSC address patterns, and whose elements are lists of commands (Listing 7.7). A command is a list starting with the command name, and followed by arguments:

- `['servo', servoNum, servoPercent]` sets servo-motor `servoNum` to the position `servoPercent`.
- `['delay', duration]` sleeps for `duration` (in seconds).
- `['complete']` waits for previous `servo` commands to complete (ie. automatically compute the delay).
- `['align']` sets the alignment point in the gesture (see subsection 7.5.3).
- `['gesture', name]` executes another gesture named 'name'.

```
gestures = {
        '/golpeBolPeque':
                [['servo', 7, 40],
                ['delay', 0.15],
                ['align'],
                ['servo', 7, 32]],

        '/golpeCymbal2':
                [['servo', 0, 49],
                ['servo', 1, 25],
                ['servo', 2, 99],
                ['delay', 0.24],
                ['align'],
                ['servo', 2, 77]],
        # ...
}
```

Listing 7.7: Robot Gesture Dictionary Example.

### 7.5.3 Latency Alignment

When the robots are played from a MIDI keyboard, network delays induce variable latency between the actuation of a key and the movement of the robot. The movement itself also incur a delay which varies depending on the robot's previous position. A skilled player can easily adapt to fairly large (e.g. a few tens of milliseconds) but constant latencies. However, variable latencies dramatically impedes their ability to play in time as well as synchronize with other musicians. Likewise, it is possible to compensate latency for pre-recorded MIDI sequences only if said latency is constant and predictible.

In order to make our system playable given variable latency constraints, incoming OSC messages were buffered and their execution delayed such that the time of impact of each gesture (specified by the `align` command) was aligned to the maximum expected latency. Despite being counter-intuitive in way, since we increased the total latency, this alignment made the total latency constant. A musician could train themselves to anticipate their

gestures so that the time of impact of the robots' sticks on their percussive instruments would match the desired target[3].

### 7.5.4 Productions

This project lead to three productions, involving different aspects of the system:

- In *Un vaso de dicha*, created at Fondation Singer-Polignac, the robot ensemble was played alone, controlled by a MIDI sequencer.
- *La Selva Virgen, en la Selva Oscura, prélude et interlude au Chant de la Terre*, was created in the Basilica of Saint-Denis. In these two pieces robots were played both by a sequencer and by a human musician, along with a human ensemble. The latency alignment mechanism described above was instrumental in providing a constant delay allowing the keyboard player to adapt its playing and synchronize with other musicians.
- The project was adapted as an autonomous installation in the hall of Fondation Singer-Polignac. The installation self-configured and played pre-programmed sequences at set times of day. Our leader-election protocol was used to select a "conductor" among the micro-controllers. This conductor ran a PureData patch that loaded MIDI sequences and translated them to OSC, and then dispatched OSC messages to the correct robots.

  Video clips from this work can be seen at the following addresses: https://youtu.be/PyJs_NrceY8 and https://youtu.be/vwnz1vBO6zg.

[3] It is worth noting that such training phase was remarkably small, as several musicians were able to play the robots orchestra in time after just a few minutes. We would like to take the opportunity afforded by this remark to thank all musicians of the ensemble Le Balcon for putting up with the constraints of our system!

# Chapter 8

# Conclusion

In this work, we highlighted the importance of human-machine temporal interaction scenarios in live shows and art installations, and the need for tools to help artists and engineers to design and play such scenarios. We presented our contribution to the tooling space, a temporal programming environment called Quadrant, built around a structure editor, a temporal language, and a symbolic-time, polytemporal scheduling model.

## 8.1 Summary

**Current Approaches And Tools.** In chapter 1, we mentioned some existing tools and identified a number of metaphors they use to convey the notion of time. We also discussed several programming interfaces, and proposed to position them on a spectrum ranging from symbolic to figurative, a distinction that we find more fruitful than the textual versus visual classification. We proposed an identification of the strengths and weaknesses of each of these approaches, and concluded on the potential of a hybrid approach coupling a symbolic temporal language with figurative user interface widgets and feedback, supported by a structure editor.

**Preliminary Work.** In chapter 2, we discussed our work on two exploratory prototypes: a textual temporal programming language, and a show controller based on hierarchical cuelists. We explored their limitations and observed that the ideas for mitigating them converged to a common, hybrid approach. On one hand, the need for introducing domain-specific user interface widgets into the textual language would lead to blurr the lines between the language

and its dedicated editor and runtime. On the other hand, breaking down the cues of the show controller into smaller, recombinable building blocks hinted at a more symbolic programming model.

**Introducing Quadrant.** In chapter 3, we introduced our temporal programming environment. We defined leading goals for its design, and gave an overview of its architecture. We presented its user-facing features, namely its structure editor, its semantic feedback, and its execution monitoring capabilities. We concluded with a number of possible user interface enhancements aimed at giving users a comprehensive view of their scenario's execution and temporality.

**Temporal Model.** In chapter 4, we presented the temporal model of Quadrant, and the scheduler supporting that model. We discussed the notion of symbolic, hierarchical timescales and time transformations, and how it has been handled in a number of previous works. We then gave a formalism for time transformations represented by tempo curves, in terms of differential equations. We considered contant and linear tempo, which have simple analytical solutions, as well as parametric tempo curves, which we solve by numerical methods. We detailed the specifics of Bézier curves, and described our implementation of piecewise tempo curves. We touched on phase synchronization and explained how we implement it using catch-up time maps built from Bézier curves. We finally presented the temporal scheduler's API and its main implementation aspects.

**Quadrant's Temporal Language.** In chapter 5, we presented Quadrant's temporal language. We gave an overview of the type system and the basic constructs of the language, and covered some more advanced features such as polymorphic procedures, modules and foreign blocks. We then presented its temporal features, which allow interacting with Quadrant's scheduler to organize computations along concurrent hierarchical time flows and control their tempo.

**Compiler and Runtime Implementation.** In chapter 6, we covered the architectures of the Quadrant's compiler pipeline and runtime, and gave some implementations details on how specific language features are implemented. We also described the execution tracking and feedback mechanisms that allows the runtime to communicate execution informations to the editor, and how the editor uses these informations to display live monitoring indicators.

**Surrounding Infrastructure.** In chapter 7, we described several components that could form the basis for a distributed temporal interaction in-

frastructure around Quadrant, allowing it to communicate and interact with external software and devices. We mentioned some core modules of the Quadrant language pertaining to networking, message passing and beat synchronization, or general language support like logging and memory management. We then presented services, that could help interoperability between Quadrant and other systems. These include a service discovery agent, a leader election protocol, and a clock synchronization protocol. We finally gave an account of an artistic production involving these distributed interaction services, and served to guide their development.

Quadrant is written in approximately 17000 lines of C code, on top of a 14000 lines platform layer and user interface toolkit called Milepost, also mostly in C (some ObjectiveC and metal shading language are used in the platform layer). Quadrant is available in a git repository here: https://forge-2.ircam.fr/fouilleul/thesis_quadrant. The source code for the components of the infrastructure described in chapter 7 and their various bindings can be accessed here: https://forge-2.ircam.fr/fouilleul/thesis_blitz.

## 8.2 Limitations and Perspectives

**Custom Interface Tokens.** Currently the only user interface token in the language is the tempo curve editor (we don't count other user interface elements such as the completion panel or the progress wheels, since they are not a defining part of the program itself). One obvious avenue for progress in the editor is to develop a collection of specialized input widgets. We can mention sliders, color pickers, piano rolls, image or video previews, or audio meters, as a few examples of useful widgets in the context of multimedia shows.

Allowing users to define their own edit-time widgets could potentially benefit to a wide number of specialized use-cases. One example is that user could define data viewers or editors for custom data formats used to communicate with other devices, such as sensors or motors. Another one is extending standard widgets provided by the environment, for example adding custom looping modes on top of a standard audio player.

In order to support user-defined widgets, the environment would have to include some kind of edit-time execution: for instance, it would need to run user-defined procedures to draw the widget's interface and react to user input, which would in turn modify program data. We should also avoid creating a strong abstraction barrier between user-defined widgets and the

rest of the language, which would hamper the use of user-defined widgets in the same way the abstraction barrier between patching and regular programming hampers the creation of user-defined boxes in most node graph environments. For this reason, we would want to provide the same language features and execution model for both edit-time and runtime parts of a the program.

These considerations hint at a even tighter integration between the runtime and the editor: essentially, the "runtime" (perhaps better called "execution engine" in this incarnation) would always be online, executing parts of the program and communicating back and forth with the editor. The editor would in turn delegate the implementation of a wide range of its features to the execution engine.

**Integrated Debugging and Profiling.** The semantic "knowledge" shared between the editor and the runtime could be leveraged more extensively to provide live debugging and profiling tools. For instance, the editor could display tooltips showing the contents of variables or the stack traces of suspended procedures on mouse-over. It could display different profiling "heat maps" or overlays over the program, surfacing information such as coverage, time spent in each block, or number of read/writes operations for each variable.

Temporal traces could be shown as live time plots similar to railways timetable graphs, along with time trajectories predicted from tempo curves and catch-up time-maps. This would give a bird eye's view of running tasks, how they branch off or join back, and the speed at which they progress. Hovering over a trace with the mouse could highlight the associated flows in the editor, and clicking a specific point of the curve could show the basic execution block that was run at that point.

The ability to record traces and replay them later can also be of great value, for example to enable partial rehearsals where some of the performers would work their part along with a partial trace of a previous execution, or with a pre-recorded trace of other performers.

**Trigger and Control.** In its current incarnation, the environment's control features are quite lacking. The only way to manually pace the execution of statements is by inserting standby instructions and resuming execution from the editor. Other than that, users have to rely on external ways of triggering actions, such as listening to incoming OSC messages in a background task.

One could for instance build a "dashboard" in PureData that would send messages to control the execution of the scenario.

Clearly, we need to match the monitoring strengths of the environment with equally useful trigger and control features, for instance:

- Launching arbitrary procedures in new tasks from the editor would be an obvious way of triggering bundles of actions.
- Manually suspending, resuming or terminating a task could also be useful, both as a safeguard and as a pacing tool.
- Manually stepping through statements one by one would implement sequential cuelists.
- Skipping suspending instructions until a specified statement could be used to enable a form of fast-forward playback to a given point the scenario.

It would also be desirable for the cursor to follow the execution of triggered tasks, unless manually repositioned elsewhere. This way, for instance, the cursor would automatically jump to the next standby instruction after the current one resumes, which would allow users to manually unfold the scenario with just one shortcut, similar to the "go" button of a show controller.

This kind of on-demand, incremental execution, also calls for an online execution engine to which code could be uploaded at a finer granularity than a whole program image.

**Developing an Ecosystem.** Quadrant was thought of as a central place where the temporal scenario of the show is specified, and which drives and reacts to all other components of the technical setup. This is akin to the notion of "centralized electronic score" developed in (Fernandez, 2021). This means that Quadrant must be able to interact with various devices and software, such as sensors, dimmers, motors, mixing and lighting desks, synthesizers, digital audio workstations, media servers, etc. As such, expanding on the primitive distributed interaction infrastructure described in chapter 7 is an important part of the effort needed to make Quadrant a useful tool.

**Live Coding.** In this work, we didn't touched on live coding. This was in part to avoid spreading the effort over too many aspects, and in part due to our own perspective as a former sound engineer (as opposed to, for example, the perspective of a digital artist). Our past personal experience is

that of someone who was in charge of designing technical setups for complex shows and operating them reliably. Live coding isn't a sustainable bet in this context, and this certainly colored our vision of Quadrant. In particular, we designed it more as a tool for technical users rather than as an instrument for digital creators.

This personal bias pushed us, at the start of the project, to favor live *monitoring* over live *coding*, and to opt for a statically typed, compiled language. Other choices of course erred more on the unsafe side, in the name of efficiency or simplicity (such as raw pointer and manual memory management), but these risks can be mitigated by debugging and testing prior to the show, whereas live coding will always be fraught with the peril of live bugs.

However, from our perspective, live coding can be extremely useful as an exploratory method, during the creation phase and the rehearsals of a show. Also, as we've seen above, there seems to be a natural development slope towards an online execution engine able to run pieces of code on demand.

Introducing live coding in Quadrant's programming model presents some challenges. In contrast to patching environments, which only expose self-contained swappable pieces of code and data, a language like Quadrant exposes data and code definitions at a much more granular level. Code expects data to follow a specific schema (in our case, a specific memory layout), and data that persists across type definition changes will be handled incorrectly. This is a problem that is faced by every hot-reloading system for compiled languages, such as Live++ (Molecular Matters, 2011). These systems either don't support data model changes, or they rely on users specifying data migration callbacks, or serializing and deserializing their data on each side of the reload point, which can be tedious and error-prone.

The problem gets even more complicated when considering hot-patching of long-lived code. Generally, live-coding systems work within the context of a run loop, where procedures are called repeatedly or in response to an event, and run immediately to completion. Thus these system generally hot patch procedures in an atomic fashion: the new version of a procedure is only used starting from the next time the procedure is entered. This isn't as useful for long-lived coroutines, that can pause and resume, and that might never be re-entered. The expectation here is that if a change occurs during a pause, the new version will be used when the coroutine resumes. This isn't as easy as it sounds, though, mainly for two reasons: first, this extends the problem of data consistency to the coroutine's local variables. Second, the

resume location might have been moved by the change, or might even have disappeared altogether!

One possible way to mitigate the risk of faulty migration code or breaking assumptions across reloads could be to run user-defined test procedures on new data and code, before committing them to the execution environment. If on of these tests failed, code and data could be reverted to their previous state and the errors could be reported in the editor. Using type introspection to generate default migration code may also lighten the burden on users in most scenarios. However, liberal use of pointers might restrict the usefulness of default migration code, or require it to trace and patch references to every piece of data that has been moved in the migration process. Restricting the use of pointers and replacing them with handles, whose underlying data can be moved without breaking references, could be a solution, at a performance and complexity cost.

Changing the code of a task that is suspended in a yielding instruction does not only only require patching the instruction pointer of the task, but potentially all return pointers along the call stack (in case a caller was edited). Matching such return points could use a simple *cell identity* heuristic, meaning a procedure which was suspended in a given cell must resume or return to the (new) next cell. If one of the call cells along the call stack doesn't exist anymore, we could just bail out and terminate the task early.

This may be an acceptable limitation, because even if we had a heuristic to match points between old and new versions of code in a more general way, it would create a "Ship of Theseus" problem: after multiple edits through invalid (non-compiling) stages, we could well reach a valid state where all parts of the code have been replaced, and trying to match the old and new versions becomes a pointless pursuit.

## 8.3 Closing Thoughts

Despite Quadrant's inherent limitations as a *prototype*, we believe the approach developed in this work contributes to the practical exploration of the problem space along two dimensions, namely temporality and programming interface design.

**Temporal Model.** The temporal model proposed here borrows the concept of hierarchical symbolic timeframes to prior works, in particular FORMULA (Anderson & Kuivila, 1990) and Antescofo (Giavitto et al., 2017). However,

it strays from these works in its handling of time transformations. Our model uses cubic Bézier tempo curves, which provide more control than common tweening functions and map directly to flexible and intuitive user interfaces. Importantly, it considers Bézier curves as a continuous abstraction, and does not rely on prior sampling and piecewise interpolation, instead introducing an adaptive numerical solver, which has better accuracy and less dead reckoning issues. We also show how synchronization to external sources can be decoupled from the concerns of score following, and how catch-up time maps can be built from Bézier curves to match both the initial and target tempos.

**Programming Interfaces.** We think this work makes a case for the hybridization of symbolic and figurative programming interfaces through the use of semi-structured program representations and the close integration of editing and runtime environments. This approach lends itself to a number of user experience improvements and allows easily representing and manipulating continuous objects such as tempo curves, without sacrificing the abstractive and combinatorial power of a symbolic programming language. It also enables novel features in live execution monitoring and debugging. This is especially important when dealing with highly concurrent and dynamic temporal scenarios, which are made possible by the temporal model proposed in this thesis.

In conclusion, we think this work both points at a large range of valuable potential features, and calls for further work, taking advantage of the insights gained during our research journey. In particular, malleability and liveliness concerns, such as live coding, may need a reassessment of some design decisions made at early stages of the project. We believe, however, that tackling those issues should be easier now that the path has been treaded once.

# References

Ableton. (n.d.). Ableton Live. https://www.ableton.com/en/live/what-is-live/

Albright, S., Leach, P. J., Gu, Y., Goland, Y. Y., & Cai, T. (1999). *Simple Service Discovery Protocol/1.0* (Internet Draft draft-cai-ssdp-v1-03). Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-cai-ssdp-v1-03

Anderson, D. P., & Kuivila, R. (1990). A system for computer music performance. *ACM Transactions on Computer Systems*, *8*(1). https://dl.acm.org/doi/10.1145/77648.77652

Apple. (2003). *Bonjour Specification* (tech. rep.). https://developer.apple.com/bonjour/printing-specification/bonjourprinting-1.2.1.pdf

Bencina, R. (2013). Oscpack. https://github.com/bobkocisko/oscpack

Berndt, A. (2011). Musical tempo curves. *ICMC*.

Blender Foundation. (1998). Blender Documentation. https://docs.blender.org/

Bouche, D., & Bresson, J. (2015). Planning and Scheduling Actions in a Computer-Aided Music Composition System. *Scheduling and Planning Applications Workshop (SPARK)*. https://hal.archives-ouvertes.fr/hal-01163284

Bray, T. (2014). The JavaScript object notation (JSON) data interchange format. https://www.rfc-editor.org/info/rfc7159

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008). *Extensible markup language (XML) 1.0 (fifth edition)*.

Bresson, J., Agon, C., & Assayag, G. (2011). OpenMusic – visual programming environment for music composition, analysis and research. *ACM MultiMedia (MM'11)*. https://hal.archives-ouvertes.fr/hal-01182394

Burnett, M. M. (1999). Visual programming. *Wiley encyclopedia of electrical and electronics engineering*. https://onlinelibrary.wiley.com/doi/abs/10.1002/047134608X.W1707

Butcher, J. C. (1987). *The numerical analysis of ordinary differential equations: Runge-kutta and general linear methods*. Wiley-Interscience.

Cash, J. R., & Karp, A. H. (1990). A variable order runge-kutta method for initial value problems with rapidly varying right-hand sides. *ACM Trans. Math. Softw.*, *16*(3). https://doi.org/10.1145/79505.79507

Celerier, J.-M., Baltazar, P., Bossut, C., Vuaille, N., Couturier, J.-M., & Desainte-Catherine, M. (2015). OSSIA: Towards a unified interface for scoring time and interaction. *TENOR 2015 - First International Conference on Technologies for Music Notation and Representation*. https://hal.archives-ouvertes.fr/hal-01245957

Cheshire, S., & Krochmal, M. (2013). *DNS-Based Service Discovery* (Request for Comments RFC 6763). Internet Engineering Task Force. https://datatracker.ietf.org/doc/rfc6763

Cockos. (2006). Reaper. https://www.reaper.fm/

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

Cont, A. (2008). ANTESCOFO: Anticipatory synchronization and control of interactive parameters in computer music. *International Computer Music Conference (ICMC)*. https://hal.inria.fr/hal-00694803

Cristian, F., & Fetzer, C. (2003). Probabilistic internal clock synchronization.

Curnow, R., & Lichvar, M. (n.d.). Chrony. https://chrony.tuxfamily.org/index.html

Cycling 74. (1997). Max/MSP. https://cycling74.com/products/max

Desain, P., & Honing, H. (1993). Tempo curves considered harmful. *Contemporary Music Review*, *7*(2). http://www.tandfonline.com/doi/abs/10.1080/07494469300640081

Dion Systems. (n.d.). Dion Systems. https://dion.systems/

Donzeau-Gouge, V., Huet, G., Kahn, G., & Lang, B. (1980). *Programming Environments Based on Structured Editors: The MENTOR Experience* (tech. rep.). https://apps.dtic.mil/sti/pdfs/ADA114990.pdf

Echeveste, J., Giavitto, J.-L., & Cont, A. (2013). *A Dynamic Timed-Language for Computer-Human Musical Interaction* (tech. rep.). INRIA.

Echeveste, J.-M. (2015). Un langage de programmation pour composer l'interaction musicale: la gestion du temps et des événements dans Antescofo.

Epic Games. (2014). Blueprints Visual Scripting. https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/

Farbrausch. (2011). Farbrausch Demo Tools. https://github.com/farbrausch/fr_public

Fernandez, J. M. (2021). *Vers un système unifié d'interaction et de synchronisation en composition électroacoustique et mixte : partitions électroniques centralisées* (Theses 2021SORUS420). Sorbonne Université. https://tel.archives-ouvertes.fr/tel-03678942

Ferris. (2017). Demotivation Notes. https://github.com/yupferris/ferris-makes-demos-notes/blob/9523886675b17ae063c7b6c307a1e47dbba3e1e7/ep-002-demotivation.md

Figure 53. (n.d.). QLab. https://qlab.app/

Fouilleul, M., Bresson, J., & Giavitto, J.-L. (2021). A Polytemporal Model for Musical Scheduling. *15th International Symposium on Computer Music Multidisciplinary Research*. https://hal.archives-ouvertes.fr/hal-03443756

Freed, A., & Schmeder, A. (2009). Features and Future of Open Sound Control Version 1.1 for NIME. https://www.nime.org/proceedings/2009/nime2009_116.pdf

Friberg, A., & Sundberg, J. (1993). Perception of just noticeable time displacement of a tone presented in a Metrical Sequence at Different Tempos. *STL-QPSR*, *34*.

Fritsch, F. N., & Carlson, R. E. (1980). Monotone Piecewise Cubic Interpolation. *SIAM Journal on Numerical Analysis*, *17*(2). http://epubs.siam.org/doi/10.1137/0717021

Giavitto, J.-L., Echeveste, J.-M., Cont, A., & Cuvillier, P. (2017). Time, timelines and temporal scopes in the antescofo DSL v1.0. *International Computer Music Conference (ICMC)*. https://hal.archives-ouvertes.fr/hal-01638115

Gibson, J. J. (1977). The theory of affordances. *Perceiving, acting, and knowing: toward an ecological psychology*. https://hal.archives-ouvertes.fr/hal-00692033

Goltz, F. (2018). Ableton Link – A technology to synchronize music software. *Proceedings of the Linux Audio Conference (LAC 2018).*
Google. (2008). Protocol Buffers Language Guide (proto2). https://developers.google.com/protocol-buffers/docs/proto
Graham, C. (2015). Geosonnix. http://www.geosonix.com/documentation/EN/gxdoc.html
Gusella, R., & Zatti, S. (1989). The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, *15*(7).
Haas, H. (1972). The influence of a single echo on the audibility of speech. *J. Audio Eng. Soc*, *20*(2). http://www.aes.org/e-lib/browse.cfm?elib=2093
Habermann, A. N., & Notkin, D. (1986). Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, *SE-12*(12). http://ieeexplore.ieee.org/document/6313007/
Halbwachs, N. (1993). *Synchronous programming of reactive systems.* Kluwer Academic Publishers.
Hall, C., Standley, T., & Hollerer, T. (2017). 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.*
Harris, S., & Sinclair, S. (2004). Liblo: lightweight OSC implementation. http://liblo.sourceforge.net
Honing, H. (2001). From Time to Time: The Representation of Timing and Tempo. *Computer Music Journal*, *25*(3). https://www.mitpressjournals.org/doi/abs/10.1162/014892601753189538
IEEE. (2008). IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. https://ieeexplore.ieee.org/document/4579760
Jack, R., Mehrabi, A., Stockman, T., & McPherson, A. (2018). *Action-sound latency and the perceived quality of digital musical instruments: Comparing professional percussionists and amateur musicians.*
Jaffe, D. (1985). Ensemble timing in computer music. *Computer Music Journal*, *9*(4).
JangaFX. (2019). EmberGen. https://jangafx.com/software/embergen/
JetBrains. (n.d.). MPS: The Domain-Specific Language Creator by JetBrains. https://www.jetbrains.com/mps/
Kim, J. L., & Belford, G. G. (1996). A Distributed Election Protocol for Unreliable Networks. *Journal of Parallel and Distributed*

*Computing*, *35*(1). https://linkinghub.elsevier.com/retrieve/pii/S0743731596900659
Lee, D. (1996). A Memory Allocator. https://gee.cs.oswego.edu/dl/html/malloc.html
Maccallum, J., Gottfried, R., Rostovtsev, I., Bresson, J., & Freed, A. (2015). Dynamic message-oriented middleware with open sound control and odot. *International computer music conference*. https://hal.archives-ouvertes.fr/hal-01165775
MacCallum, J., & Schmeder, A. (2010). Timewarp: a graphical tool for the control of polyphonic smoothly varying tempos. *International Computer Music Conference, ICMC 2010*.
Mazzola, G., & Zahorka, O. (1994). Tempo Curves Revisited: Hierarchies of Performance Fields. *Computer Music Journal*, *18*(1). https://www.jstor.org/stable/3680521?origin=crossref
McCallum, J. (2015). Libo. https://github.com/CNMAT/libo
McCartney, J. (1996). SuperCollider, a new real time synthesis language. *ICMC*.
McCurry, M. (2018). Rtosc - Realtime Safe Open Sound Control Messaging. https://lac.linuxaudio.org/2018/pdf/39-paper.pdf
Medialon Ltd. (2019). Medialon control system, user reference manual. https://medialon.com/wp-content/uploads/2019/07/M515-1-Medialon-Control-System-Manual.pdf
MIDI Manufacturers Association. (1982). The Complete MIDI 1.0 Detailed Specification. https://www.midi.org/specifications-old/item/the-midi-1-0-specification
Mills, D. L. (1988). Network Time Protocol (version 1) specification and implementation. https://www.rfc-editor.org/rfc/rfc1059
Mills, D. L. (1996). Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. https://www.rfc-editor.org/info/rfc2030
Mills, D. L. (2006). *Computer network time synchronization: The network time protocol*. CRC Press, Inc.
Mills, D. L., Martin, J., Burbank, J., & Kasch, W. (2010). Network time protocol version 4: Protocol and algorithms specification. https://www.rfc-editor.org/info/rfc5905
Molecular Matters. (2011). Live++. https://liveplusplus.tech/index.html
Moon, D. (2022). Tylr. https://github.com/hazelgrove/tylr
NetBIOS Working Group. (1987). *Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods* (Request for Comments RFC 1001). Internet Engineering Task Force. https://datatracker.ietf.org/doc/rfc1001

Norman, D. A. (1988). *The design of everyday things* (Revised and expanded edition). Basic Books.

OASIS. (2009). OASIS Web Services Dynamic Discovery (WS-Discovery) Version 1.1. http://docs.oasis-open.org/ws-dd/discovery/1.1/wsdd-discovery-1.1-spec.html

Omar, C., Voysey, I., Hilton, M., Aldrich, J., & Hammer, M. A. (2017). Hazelnut: A Bidirectionally Typed Structure Editor Calculus. *ACM SIGPLAN Notices*, *52*(1). http://arxiv.org/abs/1607.04180

Open Management Group. (2018). *Interface Definition Language* (tech. rep.).

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). Integration of Ordinary Differential Equations. *Numerical recipes in c (2nd ed.): The art of scientific computing.*

Puckette, M. S. (1996). Pure Data. https://puredata.info/

Roberts, C., Wakefield, G., & Wright, M. (2017). 2013: The Web Browser as Synthesizer and Interface. *A NIME Reader.* http://link.springer.com/10.1007/978-3-319-47214-0_28

Roberts, C., Wright, M., Kuchera-Morin, J., & Höllerer, T. (2014). Gibber: Abstractions for Creative Multimedia Programming. *Proceedings of the 22nd ACM International Conference on Multimedia.* https://dl.acm.org/doi/10.1145/2647868.2654949

Schnell, N., Saiz, V., Barkati, K., & Goldszmidt, S. (2015). Of time engines and masters an API for scheduling and synchronizing the generation and playback of event sequences and media streams for the web audio API. *WAC.* https://hal.archives-ouvertes.fr/hal-01256952

Scratch Foundation. (2003). Scratch. https://scratch.mit.edu/

Smode Tech. (n.d.). Smode. https://smode.fr

Steinberg. (n.d.). Cubase. https://new.steinberg.net/cubase/

Teitelbaum, T., & Reps, T. (1981). The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, *24*(9). https://dl.acm.org/doi/10.1145/358746.358755

Von Hanxleden, R., Bourke, T., & Girault, A. (2017). Real-time ticks for synchronous programming. *2017 Forum on Specification and Design Languages (FDL).* http://ieeexplore.ieee.org/document/8303893/

Wang, G., Cook, P. R., & Salazar, S. (2015). ChucK: A Strongly Timed Computer Music Language. *Computer Music Journal*, *39*(4). https://www.mitpressjournals.org/doi/abs/10.1162/COMJ_a_00324

Wright, M. (2002). Open Sound Control Specification 1.0. https://opensoundcontrol.stanford.edu/spec-1_0.html