Multimedia Interaction with NTCC

Salim Perchy

Multimedia Interaction with NTCC

Author:

Salim Perchy

Advisor:

Dr. Camilo Rueda

Universidad Javeriana

Santiago de Cali

September 25, 2013

MULTIMEDIA INTERACTION WITH NTCC

YAMIL SALIM PERCHY
Autor

Nota de Aceptación:
Certificamos que el presente Trabajo de Grado satisface, en alcance y calidad, todos los requisitos que demanda un Trabajo de Grado de Maestría.

CAMILO RUEDA Ph.D
Director

NÉSTOR CATAÑO Ph.D
Jurado

GERARDO SARRIA Ph.D
Jurado

Aprobado en cumplimiento de los requisitos exigidos por la Pontifica Universidad Javeriana - Cali, para optar por el título de Magister en Ingeniería.

MAURICIO JARAMILLO AYERBE Ph.D
Decano Académico de la Facultad de Ingeniería

WILLIAM ANDRÉS OCAMPO DUQUE Ph.D
Director de Posgrados de Ingeniería

Resumen

Los fenómenos musicales son el subconjunto más estudiado de las bellas artes actualmente. Desde formalizaciones matemáticas, análisis psicológicos hasta modelos computacionales son algunas de los caminos que la comunidad científica ha tomado para explorar el campo musical. Uno de estos formalismos aplicados a modelar música se llama `NTCC` , un cálculo de procesos basada en restricciones donde los operadores son parte de programas que pueden comportarse como agentes musicales.

Este trabajo tiene como objetivo proveer una herramienta de trabajo para la interacción, construcción y deducción de propiedades musicales de una interpretación con base en una partitura conocida a priori. El método está dividido en 2 partes, una construcción interactiva de un proceso `NTCC` de eventos musicales ejecutados a través del programa *ANTESCOFO* y un módulo *model checker* para la salida de la parte anterior. Un esquema de comunicación entre ambas partes fue implementado y documentado y, finalmente, un ejemplo con una pieza musical es mostrado en conjunto con un análisis de sus resultados.

Abstract

Musical phenomena are the most formally studied subset of the fine arts in our present time. From mathematical formalization to psychological analysis onto computer models are among the traces the scientific community have taken to explore music. One such formalism applied to model music is called `NTCC` , a constraint-based process algebra where operators are part of programs that may behave as musical agents.

This work is aimed to provide a solid framework for interacting, constructing and deducting musical properties from an interpretation of a previously known score. The method is divided into two parts, an interactive construction of an `NTCC` process from executed musical events through the program *ANTESCOFO* and a model checking module for the output. A communication scheme between the two parts is implemented and documented, and finally an example with a musical piece is given along with analysis of its results.

Contents

Abstract

Table of Contents **i**

List of Figures **iii**

List of Tables **iv**

List of Algorithms **v**

Acknowledgement **vi**

1 Introduction & Motivation **1**
1.1 Document Structure 2

2 Concurrency Theory and Computer Music Background **3**
2.1 Process Calculi 3
2.1.1 Constraint Programming 5
2.2 Computer Music 8
2.2.1 Automatic Score Followers 9
2.3 Multimedia Protocols 13
2.3.1 OSC(Open Sound Protocol) 13
2.3.2 Minuit Protocol 16
2.4 Property verification in Timed Systems 17
2.4.1 CLTL 19

3 Project `React+` and Problem Overview **22**
3.1 Previous Work 22
3.1.1 `REACT` 22
3.1.2 `REACT+` 23
3.2 Problem Statement 24
3.2.1 Definition 24
3.2.2 Application 24
3.2.3 Goals 24

3.2.4 Scope 25

4 Musical Property Verification **26**
4.1 Design 26
4.1.1 Communication Strategy 27
4.1.2 Musical translation to NTCC 28
4.1.3 Property Checking Technique 29
4.2 Implementation 30
4.2.1 OSC and *Minuit* Protocol 30
4.2.2 Model Checking 32
4.2.3 Searching the property satisfaction 38

5 Results and Conclusions **40**
5.1 Tests and Results 40
5.1.1 The Piece 40
5.1.2 Musical Properties 41
5.2 Discussion and Contributions 47
5.2.1 Conclusions 48
5.2.2 Future Work 49

Bibliography **50**

Appendices **56**

A Framework Tools **57**
A.1 ANTESCOFO Pure Data Syntax 57
A.2 Minuit Example 57

B Multimedia Protocols Documentation **61**
B.1 OSC-0z 61
B.2 *Minuit* implementation 67
B.2.1 Minuit for Pure Data 72

C Model Checking Documentation **75**
C.1 ntccModelChecker 75

D ANTESCOFO Scores **89**
D.1 First property score 89
D.2 Second property score 92
D.3 Third property score 95

List of Figures

1.1 Marble level in Mario Galaxy(©Nintendo) 2

2.1 NTCC Syntax 7
2.2 Score Follower Design 9
2.3 ANTESCOFO architecture[Con10] 10
2.4 ANTESCOFO abstract syntax 11
2.5 ANTESCOFO appearance and event language 11
2.6 ANTESCOFO synchronization strategies[Ech+12] 12
2.7 OSC Scheme 14
2.8 Model Checking scheme 18
2.9 **CLTL** Syntax 20

4.1 Overall design scheme of automatic musical property verification . 27
4.2 OSC-Oz data-flow 31
4.3 NTCC state the kripke structure 35
4.4 Krikpe structure for the NTCC process 36
4.5 Unobservable state reductions 37
4.6 Reduced Kripke structure for the NTCC process 37

5.1 Interval notes of a bar in red color 42
5.2 Rhythmic cell and its resolutions in red color 43
5.3 On-beat notes marked in color red 46

A.1 *Minuit* namespace example 59

B.1 Minuit Client patch for PD 73
B.2 Minuit Client 74

List of Tables

A.1 ANTESCOFO concrete score language 58

List of Algorithms

1 Main Model Checking Algorithm 32

Acknowledgement

This work, coordinated by Camilo Rueda PhD[1], is subscribed to the research group *AVISPA*[2] and is part of the project `REACT+`[3] which is funded by *Colciencias*[4], `REACT+` is a continuation of the project `REACT`[5] also funded by *Colciencias*.

AVISPA is based in Pontificia Universidad Javeriana - Seccional Cali[6] and Universidad del Valle[7] with further collaboration from the IRCAM[8](Institut de Recherche et Coordination Acoustique/Musique)[9] and the INRIA team Comète[10] based in the Lix Laboratory[11](Laboratoire d'Informatique de L'École Polytechnique)[12]

The author wishes to express his deepest regards and gratitude to his mentor Camilo Rueda, without him this work, and the master degree the author is aspiring to, would still be a fuzzy prospect. The author can only repay by passing onto

[1] http://cic.puj.edu.co/~crueda/camilo/Home.html
[2] http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:avispa
[3] http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:react-plus
[4] http://www.colciencias.gov.co/
[5] http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:react
[6] http://www.javerianacali.edu.co
[7] http://www.univalle.edu.co/
[8] http://www.ircam.fr/
[9] Institute of Musical/Acoustical Research and Coordination
[10] http://www.lix.polytechnique.fr/comete/
[11] http://www.lix.polytechnique.fr/
[12] Informatics Laboratory of the Polytechnic School

him good films and good musical discussions.

This work is dedicated to the author's parents; Yamil Perchy and María Janeth Bocanegra and his soul-mate Laura Isidro, whose advice, wisdom, humility and care have made the author what he is today.

Chapter 1

Introduction & Motivation

One, of the six classical arts commonly called *fine arts* manages to, more often than not, insert itself in every healthy conversation about the relationship between mathematics and creativity. This topic is not relegated to mere chat, even the book *Gödel, Escher, Bach*[Hof79] is entirely devoted to discuss mathematics into the process of creation.

This one art, called music, up to this present day still tides waves in scientists to promote a plethora of machine assisted tasks such as: composition, classification, aided or automatic execution, identification, feature extraction, data mining, persistence, orchestration, improvisation, instrument innovation and cognition to name some.

Computer Music is the field in charge of evolving the role of computers in musical environments, and as mature as it is now, it yet has to evolve significantly because of the several dimensions the field encircles. *Music Information Retrieval* or MIR for short is a sub-field of Computer Music where its goal is to extract relevant information from a musical input. The form of the musical input, a virtual representation of an audio signal or a symbolic representation of a musical piece, greatly divides this sub-field in the methods applied to get results.

Symbolic representations, a higher level of abstraction of audio, have recently been in a kind of stockade, because of their lack of genericity and robustness compared to audio signal representations. However, this work aims to diminish this by using symbolic representation to extract and prove features of tonal music in the form of logical properties, properties where its value is true or false.

Musical property verification has applications on scenarios where the desired outcome must fulfill rules or have some well defined characteristics, for example producing music with certain assured features to comply with other ideas or connect with other outcomes.

Figure 1.1: Marble level in Mario Galaxy(©Nintendo)

A particular example can be traced to the video game *Super Mario Galaxy* [Nin07]. On certain parts of the game, the main character *Mario* has to advance through the world on top of a big marble as shown in figure 1.1. Because the world is fully three dimensional, the ball complies with gravity, friction and other physical properties, the background music is programmed to keep the same tempo as the velocity of the marble displacement[1]. This of course, carries properties of musical speed to comply with correct visual feedback and musical sampling to comply with audio consistence.

Scenarios similar to this motivate the work presented in this document, and the author hopes it contributes towards a more conciliatory view of the MIR domain and the symbolic representation of music, and a gap reduction of computing and music.

1.1 Document Structure

The document is divided as follows: In chapter 1, the author writes a motivation of the whole work and asserts certain applications of the project. Chapter 2 revisits and establishes the needed theoretical background for the concepts, methods, frameworks and tools used in this work. Later, chapter 3 introduces the `REACT+` project and states the main and secondary objectives as well as the project scope. In chapter 4, the main solution method is designed and explained, and details of every module and technique used in it are presented. Lastly, chapter 5 presents some tests of the proposed scheme along with explanation of their results and offers concluding remarks, contributions and future work advises.

[1] A video of this feature is at `http://www.youtube.com/watch?v=T2xubimrw24`

Chapter 2

Concurrency Theory and Computer Music Background

This chapter is dedicated to the purpose of giving the reader a comprehensible theoretical framework. It covers the necessary subjects required to understand the solutions and programs developed in this document.

Because of the inter-disciplinary nature of this work, a complete state of the art of the various fields treated here is a task of lengths longer than necessary. Instead this chapter focuses only on the specific topics related to the work as a whole and it is a complete and self-contained reference to the theory and practices used and/or relayed on.

The framework describes three specific fields in computer science; in section 2.1 an explanation of process calculi is given along with their formal background, specific calculi used in this work is also explained in depth, section 2.2 details the field of computer music and specifically the topic of automatic score followers.

Section 2.3 refers to the protocols designed to communicate processes dealing with multimedia data and finally, section 2.4 explains the techniques and theory behind verifying properties in timed systems and particularly the `NTCC` calculus.

2.1 Process Calculi

Process calculi(also referred to as process algebras) were created out of the necessity of formalizing the concept of computation in concurrent processes. A computation step is defined as an action taken to achieve a goal(or state) in any machine, alternatively it can also be defined as a mathematical function that,

depending on its parameters(inputs) changes or evolves into something(outputs). An ordered set computation steps conceptualize the process of computation.

This view led mathematicians Alonzo Church and Alan Turing to create the λ-calculus and the *Turing Machine* in the decade of 1930. Both are definitions of a computational machine and were made with a formal mathematical language so as to provide a rigor, in the sense of correctness, to any computation made within these machines. With this notion, a program written specifically for them can be checked for correctness and other properties which may be desirable for the program to comply, for example a calculator that never tries to perform division-by-zero.

With the advent and demand of faster and powerful machines, multicore and multiprocessor computers are the current driving force in hardware architecture design. In this case parallel computation takes place as the principal paradigm selected to solve original as well as new computational problems using concurrent processes. Theoretical machines like the Turing Machine are no longer convenient concepts to evolve onto concurrency as they treat computing as *sequential* steps.

Process algebras are born to address this formalization, in this sense they can be described as a set of mathematical operators, each one modelling common concurrent operations. When a level of mathematical rigor is required in order to preserve correctness and other properties, these calculi are defined with *Operational Semantics*[Plo81] and *Denotational Semantics*, the former defining what each operator **does** and the latter what each operator **means**[Mit96]. There exist more types of semantics and these two are the most common.

Mathematical models for concurrent computing have existed as far back as 1973 where the Actor Model was originated in [HBS73], of great relevance is the work of Robin Milner with his *Calculus of Communicating Systems* introduced in [Mil80], this calculus sets up an invisible and coherent way of passing messages between concurrent processes. The reader can find a comprehensive description of all operators of this calculus in [AG05].

Later on, Robin Milner again introduced the π-calculus in [Mil99] as an improvement over its predecessor *CCS*. Here, a greater flexibility over the model of communication allows for passing the identities of the communication channels in order to provide the calculus with dynamic intercommunication schemes. This calculus is intended to be the concurrent equivalent of the λ-calculus in terms of expressiveness and simplicity.

The π-calculus has served as a basis of several variations tackling specific domains of problems in the computer science field; examples of this are the Spi-Calculus [AG99] which deals with cryptographic communication protocols and the Ambient-Calculus [CG00] which models mobile computational ambients.

2.1.1 Constraint Programming

Constraint Satisfaction Problems(CSP for short) are a narrow but significant set of mathematical problems where their definition is expressed as a series of limitations and/or properties(constraints) the solution to the problem must comply with. They are characterized for having a space of possible solutions where their *search* has a high price in terms of time complexity. The game *Sudoku* is a popular example of a CSP problem, inherently these problems differentiate themselves by the fact that they are specified in terms of their solution and not the problem itself [ch.12][RH04].

Mathematically speaking, a CSP can be defined as the triplet $\langle X, D, C\rangle$ where X is a set of variables each describing a characteristic of the solution to the problem, D is a set of domains of each variable in X respectively and C is a set of constraints. Each of these constraints is defined as a pair $\langle x, c\rangle$ where x is a subset of X and c is a relation of the variables in x. v is a solution of the problem if and only if it is a mapping(valuation) of the variables into the domain $(v : X \longrightarrow D)$ that satisfies every constraint in C, satisfaction of a constraint meaning $c(v(x))$ holds [RBW06].

Incorporating CSPs into concurrent programming leads to a new family of process algebras commonly called the `cc` languages(short for Concurrent Constraint). The initial effort is due in [SRP91] with the Concurrent Constraint Programming(`CCP`) calculus, in here *agents*(programs) concurrently compute over a store of values, these values are no longer a single valuation of variables but rather a constraint system. Agents can now *ask* if a constraint is entailed(holds) or *tell*(add) information to the store with a constraint.

These `CCP`-type languages are parameterized by the concepts of *asking* and *telling* constraints allowing for more flexibility. How they are performed is ultimately a *constraint system*, a precise mathematical definition of how these two actions are carried out. The syntax for `CCP` is given in [GJS02] as:

$$P, Q := \ulcorner c \mid c \rightarrow P \mid \exists x.P \mid P \wedge Q \lrcorner$$

Intuitively, a process P may impose a constraint c, such constraint could be a formula over variable(s) consistent with the actual constraint system. Also a process may ask if constraint c holds and launch process P, in here if there is no sufficient information to deny nor imply the constraint c the process will stall until more information is available. A process P may contain a local variable x only visible to it. Finally, a process can be the parallel composition of processes P and Q.

To give a simple example on a constraint system, one can use the propositional

logic and tell the constraints $a > 5$ and $b < a$, whereas if one asks for the constraint $b = 10$ it cannot be entailed. A constraint system can also be adjusted or expanded into more complex scenarios, the reader may refer to [Kni+12] for examples of this.

As its maturity progressed, more features were added to this initial work. The concept of time was added in two works; TCC[SJG95] and TCCP[AGV06]. These extensions were introduced as means of reasoning about time in a process execution line, more specifically TCC models the concept of *time units* for dividing the execution in discrete intervals.

Later on, Frank Valencia introduced NTCC which further extends TCC with the concept of *non-deterministic* behaviour and *asynchrony* [Val02]. An explanation of this calculus in detail is in section 2.1.1 as it is the base programming language of the REACT+ project.

More advanced extensions were also introduced to the cc framework. RTCC[Sar08] has a slightly different concept of time from NTCC *time units* exist but they have a fixed duration. It also adds two real-time operators; *delay* a process and amount of time and *strong preempt* the execution of a process if a constraint can be entailed in such execution.

Other extensions include; PNTCC [PR08a] adding probabilistic operators to NTCC and UTCC [OPV07] extending NTCC with process communication a la *CSS*.

There is also written software to run, simulate or interpret programs specified in concurrent computing models. The language *erlang*[Lab87] is among the languages who supply a natural syntax for parallel computing, also X-10[Cha+05] is a java-like language aimed to provide full-parallelism. *erlang* is developed by *Ericsson* and X-10 is designed by the creator of CCP in *IBM*.

Software for solving CSP's are either libraries to another language or standalone programming languages. Gecode[Tac08] provides a constraint solving library for C++. The language *Mozart*, an implementation of the *Oz* specification contributes a framework for working constraint satisfaction problems.

NTCC (Non-Deterministic Timed Concurrent Constraint programming)

The NTCC language is an extension of the TCC calculus, which in turn is also an extension of the original CCP calculus. It is the principal programming paradigm in the project REACT+(see chapter 3) and also the base language for which this work is tailored. As NTCC has features of great relevance this small section is dedicated to describe it in detail.

As with CCP, the computational *agents* here are processes that ultimately either impose to or validate a constraint from a store. The basic agents of NTCC are

$$
\begin{array}{lcll}
P, Q & := & \textbf{tell}(c) & (2.1)\\
 & | & P + Q & (2.2)\\
 & | & \textbf{unless } c \textbf{ next } P & (2.3)\\
 & | & \textbf{when } c \textbf{ do } P & (2.4)\\
 & | & \textbf{next } P & (2.5)\\
 & | & \star P & (2.6)\\
 & | & \textbf{local } x_1[, \ldots, x_n] \textbf{ in } P & (2.7)\\
 & | & P \parallel Q & (2.8)\\
 & | & ! P & (2.9)
\end{array}
$$

Figure 2.1: NTCC Syntax

closure operators over the store, meaning it complies three properties: (1) *monotonic*(i.e. each time an agent imposes a constraint to a store there is at least the same or more information), (2) *increasing*(i.e. if store s_1 is contained in store s_2 then imposing the same constraint on both stores preserves this containment relation) and (3) *idempotent*(i.e. agents imposing the same constraint more than once have the same result) .

The *monotonic* property assures the store is always growing with new information as more constraints are imposed, and this may bring the scenario where the store becomes *inconsistent* and anything can be deduced from it. This state is to be avoided and is parallel to an exception or a segmentation fault.

In NTCC time is *discrete* but not *uniform*, this means it is divided into *time units* but they may not be of the same length. Also, in each *time unit* the set of current executing processes are given time to evolve(compute) into a resting point(no more computations left), this marks the end of said unit. After this a new time unit begins with new processes either added explicitly by the program or residues(carried over by processes of past time unit) and an emptied store.

Figure 2.1 shows the basic agents and syntax of NTCC, operators 2.7, 2.8 and 2.9 (locality, parallelism and replication) are primitives commonly found in process algebras. Locality establishes a variable **only** visible to process P, and replication spams P in all time units that follow.

Operator 2.5 postpones the execution of P to the next time unit, P may be called a *residual process* from the actual time unit to the next. Operator 2.6, called *asynchrony*, postpones the execution of P indefinitely but eventually will execute. With the concept of 2.5 the replication 2.9 can be seen as:

$$! P \equiv P \parallel (\textbf{next} ! P)$$

Operator 2.1 simply imposes a constraint c to the current store. . The non-determinism operator 2.2 chooses to execute non-determinedly P or Q but not both. With non-determinism, operator 2.6 can be re-defined as:

$$\star P \equiv P + (\textbf{next} \star P)$$

Operator 2.3 constantly checks the store for c in the current time unit, if not found it then promotes P to the next time unit. The choice operator 2.4 is relevant because it somehow reflects conditionals; in the current time unit P is stalled until c can be deduced from the store. Operator 2.4 together with operator 2.2 can emulate a *switch*-like conditional:

$$\sum_{i \in I} \textbf{when } c_i \textbf{ do } P_i$$

All P_i are stalled until one (or more) of the guards c_i can be deduced from the store, it then chooses non-determinedly one P whose guard can be deduced and the others P_i are discarded.

2.2 Computer Music

Applying computational theory to solve problems in the field of musicology is called *Computer Music*, the specific set of musical problems within the grasp of computer science concern decision-making and automation [Maz03].

Computer assisted composition also called *algorithmic composition* relates to the task of creating a musical piece totally or partially. The techniques used here include grammars, learning algorithms, mathematical models or knowledge-based constructions[Xen01]. Another similar problem called *machine improvisation* concerns the automatic musical improvisation generation, it can be offline(feeding the parameters a priory) or online(doing real-time improvisation) [RAD06].

Computer performance refers to the automatic execution of musical instruments according to a musical score, *automatic orchestration* deals with the embellishment of musical lines, *interactive music* is the field dealing with human-computer musical interaction and *musical representation* tackles the problem of virtual representation and rendering of musical sounds and symbols.

A current trend is the field of *music information retrieval*(MIR for short), here

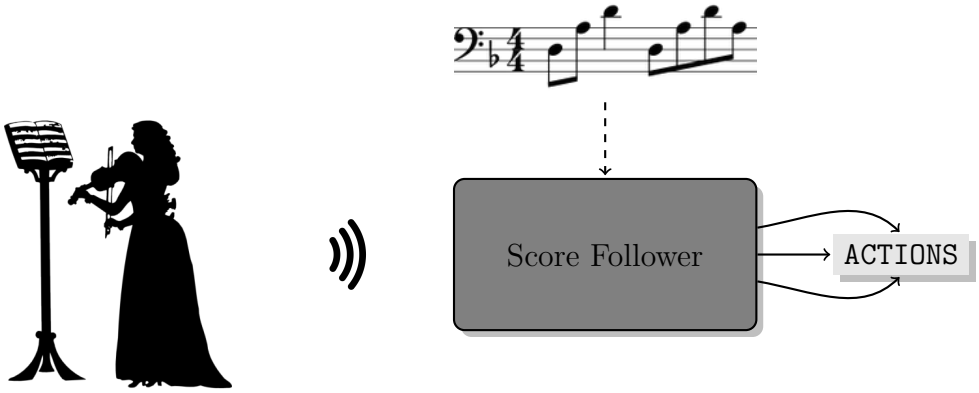


Figure 2.2: Score Follower Design

a more low-level approach of deducing information from a musical acoustic signal is the prime strategy. Common goals include; song-author-genre recognition, metric recognition, feature tagging and ontological classification.

2.2.1 Automatic Score Followers

Of relevance to this work is the field of computer music called *Automatic Score Following*. A score follower is basically an alignment of an audio signal to a symbolic representation of such signal(see figure 2.2), this is done in real-time and this alignment permits several actions upon reaching a point in the score, more importantly orchestration. For example, a *piano concerto* consist of a piano soloist accompanied by an orchestra, the soloist could rehearse his part alone without the presence of the full orchestra [Con11]

Automatic alignment is a complex problem that involves several variable environment characteristic like noise, execution speed, instrument tuning, error handling and synchronization among others. Common techniques to tackle these issues are pitch detection with string matching techniques, grammatical inference, dynamic time warping and hidden markov models [Con10].

ANTESCOFO(ANTIcipatory SCOre FOllowing system)

The ANTESCOFO system is an efficient program for audio-signal to score alignment that aims to, in real-time correctly position in a score the performance of a musician playing an electronic or acoustic instrument [Con08b]. It is a multimedia application inside the computer music field and a significant program to perform multimedia interaction in real time(see chapter 3). This section explains in depth the relevant aspects of the system used and treated withing this work.

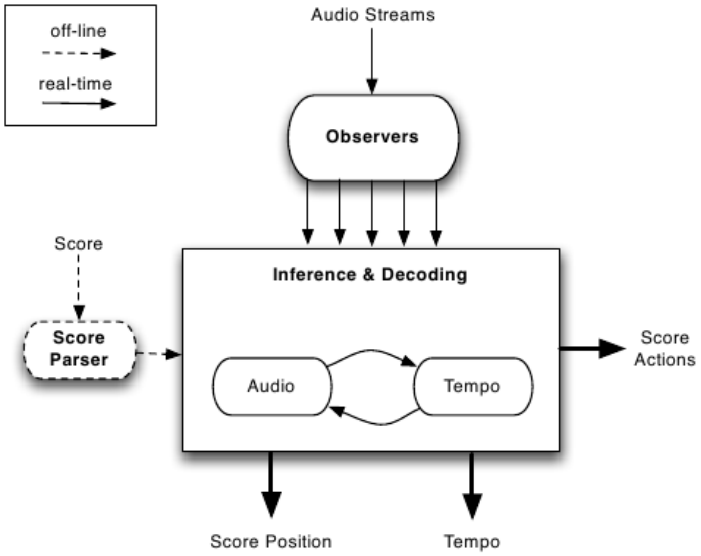


Figure 2.3: ANTESCOFO architecture[Con10]

ANTESCOFO's architecture is depicted in figure 2.3. Intuitively, it consumes a musical score *a priory*(offline) through a score parser translating it into an internal structure. In real-time(online) an audio stream is fed to the system through a filter taking the important signal features. Internally, ANTESCOFO goes through an infinite cycle (at least until the signal is gone) of inference and decoding, all this while accurately outputting the current music tempo and score position [Con08a].

Alternatively, the score given *a priory* may contain a number of actions aligned to events in the score as pictured in figure 2.5b. These actions have to be correctly synchronized under various circumstances like missed or delayed events. The figure 2.4 describes the abstract syntax of the internal language; rule 2.10 shows that a score can be empty, have events, groups, loops or continuous groups, except for score events the other components may have an associated delay s expressed in beats or absolute time(milliseconds). [Ech+12]

Rule 2.11 makes score events with duration c, these will be discussed later, rule 2.12 makes a group of one or more actions with a label ℓ, a synchronization strategy *synchro* and an error handling strategy *error* , rule 2.13 makes a group of actions repeat n times each period p(in beats or milliseconds), note that repetitions may overlap. Rule 2.14 shows any group or loop can be nested as well as have individual actions, rule 2.15 linearly interpolates the values of vector v. These values are sampled at a rate *step* to achieve perceptual continuity and in an ordered manner(from values v_{i-1} to the values of vector v_i), this interpolation lasts the time interval $\left[t_0 + \sum_0^{i-1} d, t_0 + \sum_0^{i} d\right]$. where t_0 is the initial time of

$$score := \epsilon \mid event\ score \mid (d\ group)\ score \mid (d\ loop)\ score \mid (d\ cont)\ score \tag{2.10}$$

$$event := (e\ c) \tag{2.11}$$

$$group := \textbf{group}\ \ell\ synchro\ error\ (d\ action)+ \tag{2.12}$$

$$loop := \textbf{loop}\ \ell\ synchro\ error\ p\ n\ (d\ action)+ \tag{2.13}$$

$$action := a \mid group \mid loop \mid cont \tag{2.14}$$

$$cont := \textbf{cont}\ \ell\ step\ error(d\ \mathbf{v})+ \tag{2.15}$$

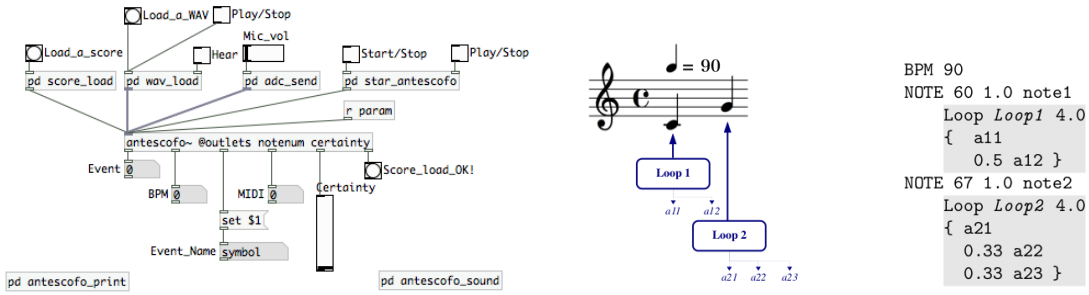
$$synchro := \textbf{loose} \mid \textbf{tight} \tag{2.16}$$

$$error := \textbf{local} \mid \textbf{global} \tag{2.17}$$

Figure 2.4: ANTESCOFO abstract syntax

interpolation. Finally rules 2.16 and 2.17 establish the possible synchronization and error strategies respectively [Ech+12].

ANTESCOFO achieves an efficient audio-to-score alignment by using an anticipatory system. This system is based on two agents; an audio agent in charge of audio recognition and a tempo agent estimating current score position and tempo, each one receives feedback from the other as well as make decisions based on their opposite's predictions. This collaborative scheme is always updating itself as new real audio events come. The audio agent uses a theoretical framework of *Hidden Hybrid Markov/Semi-Markov Time Occupancy* mathematical models for effectively positioning an event in the score, on the other hand the tempo agent uses an stochastic model of time based on an *Attentional Model of Tempo*. All these strategies are carefully crafted to preserve a low execution time in order to achieve real-time responses, the reader may refer to [Con10] to an in-depth explanation of these anticipatory system.



(a) Usage in Pure Data (b) Action language[Ech11]

Figure 2.5: ANTESCOFO appearance and event language

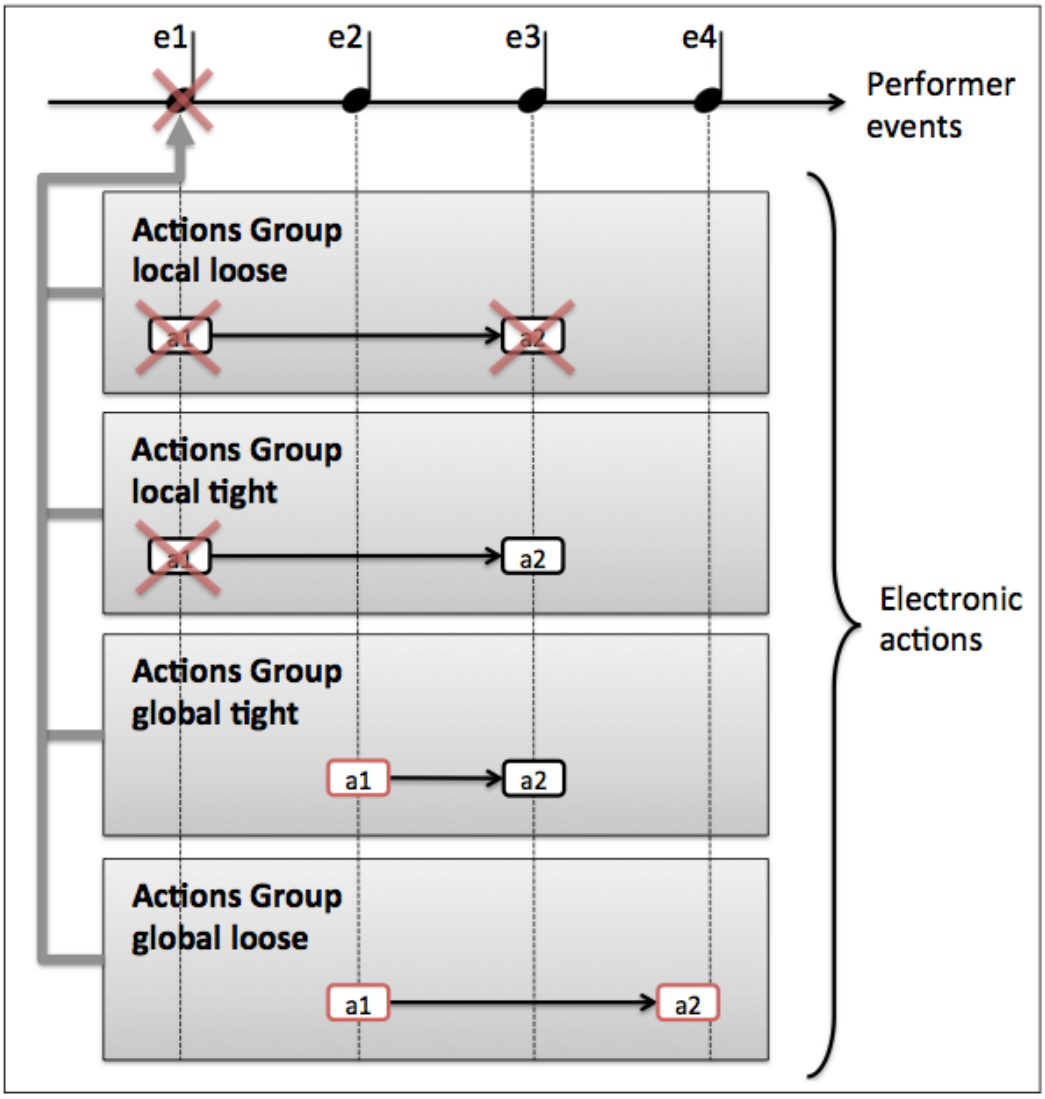


Figure 2.6: ANTESCOFO synchronization strategies[Ech+12]

The automatic online recognition of music played by humans in real life is not always a deterministic output problem, as humans are also bound to make mistakes, change mood, dynamics and speed and even the acoustics, noise and characteristics of the environment make the sound always different. ANTESCOFO deals with these with synchronization and error strategies attached to an action, group or loop; in case of a missed triggering event a **local** error strategy dismisses the action and a **global** strategy launches the action immediately upon recognition of absenteeism of this event. In the case of a change of tempo by the musicians a **loose** synchronization strategy reschedules the actions with respect to this real-time changes and a **tight** synchronization strategy reschedules the actions relative to their nearest original triggering event. Figure 2.6 shows the output of combining these strategies in case of missed events, the reader may find an in-depth explanation of these strategies in [Ech+12].

The implementation of ANTESCOFO used in this project was programmed in the visual language *Pure Data*, see figure 2.5a. The actual score syntax slightly varies from the abstract syntax presented above, appendix A.1 shows each possible command with an example and a description for each example.

2.3 Multimedia Protocols

In order to communicate applications or frameworks over external channels and preserve compatibility with other or future applications, the need arises to rely on established rules of communication. Network protocols are created for this purpose among others like security and low-latency. A multimedia protocol is a normal communication protocol in essence but it is tailored to multimedia data streaming, because of this other properties like low bandwidth consumption, appropriate representation of multimedia data structures and simplicity have more priority.

In such protocols, a fit abstraction of sent and received data is an important feature that marks the future community adoption. A pioneer of multimedia protocols was the MIDI(Musical Instrument Digital Interface) specification [Ass82], it allowed electronic instrument synchronization/communication and tonal music representation with low data throughput. It has been a relevant protocol in the electronic music scene for decades.

2.3.1 OSC(Open Sound Protocol)

With the increased availability of more computer power and demand of better quality in multimedia presentation, MIDI is unfit to improve audio quality due to

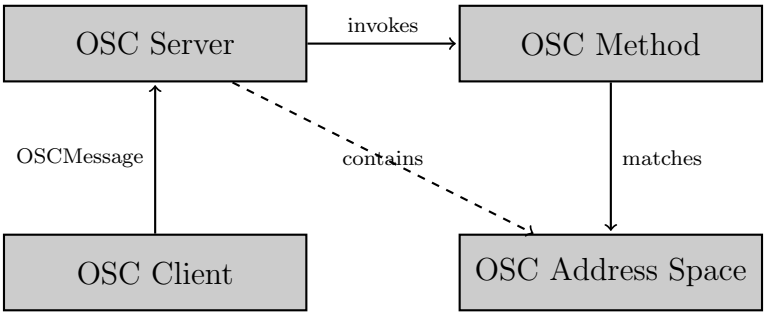


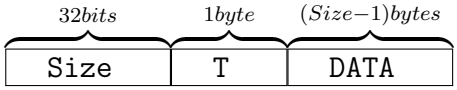
Figure 2.7: OSC Scheme

its design and low-level specification(i.e. byte ordering, chunks, etc).

A more expressive and simple protocol called OSC(Open Sound Control) was introduced in 1994 by the University of California[WFM03]. Unlike Midi, this protocol allows for more flexible message construction, a more suited value passing scheme and a wider range of multimedia applications than just music applications as MIDI was.

The general scheme of OSC, exposed in figure 2.7 is a client-server protocol, the *OSC Client* sends a unit of transmission called an *OSC Packet* to an *OSC Server*. The server then invokes the corresponding *OSC Method* implemented in the server and referenced in a *OSC Address Space*.The process of deducing what method is to be executed is called *dispatching* and follows a unix-like pattern matching base on the *OSC Address* inside the *OSC Messages*.

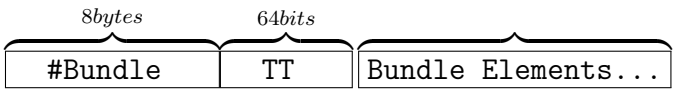
An *OSC Packet* is of the form:



where:

Size: Size of package.
T: Type of package, can be `/` or `#`.
  `/`: Means a normal *OSC Message*
  `#`: Means an *OSC Bundle*
DATA: Package data.

An *OSC Bundle* is a contention-recursive package(i.e. a package containing packages), it contains an *OSC TimeTag* `TT` specifying the execution time of the contained elements:



where:

#Bundle: *OSC String #Bundle*

TT: Time tag of the form: $\overbrace{\texttt{Seconds}}^{32bits}\overbrace{\texttt{Frames}}^{32bits}$
- `Seconds:` Seconds since *January 1 1900*
- `Frames:` 1 frame = 200ps (1ps = 1×10^{-12}seconds)

Where `Seconds = 0` and `Frames = 1` means *immediately*

Bundle Elements: Packages in bundle

An *OSC Message* is structured as follows:

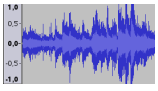
`Address Pattern String`	`,Type Tag String`	`Arguments...`

Address Pattern: */OSC Address*

,Type Tag String: Types of arguments, separated by a comma from. They can be:
- `i:` 32bit integer number
- `f:` 32bit IEEE 754 floating point number
- `s:` Null-terminated string(its size is always multiple of 4)
- `b:` *OSC blob*, a 32bit size count followed by the binary data(its size is also always multiple of 4)

Arguments: Actual data of the arguments specified in the Type Tag String field.

Some hypothetical examples are:

$$/orquesta/strings/violin1/play,\ f\ 440.0$$
$$/body/arms/right/hand/thumb/rise,$$
$$/ensemble/sequencer/playwave,\ b\ \text{  }$$
$$/orquesta/strings/*/playchord,\ s\ \texttt{a\_minor}$$
$$/orquesta/strings/violin[1-5]/play,\ f\ 220.0$$
$$/orquesta/winds/\{basson, horn\}/rest,$$

Several extensions to provide better pattern matching, user-composed time-tags, and other types of arguments have been proposed to OSC resulting in version 1.1. The version 2.0 is a work in progress [FS09].

2.3.2 Minuit Protocol

The OSC multimedia protocol has gained momentum to be considered the *de facto* standard in multimedia interaction and its expressiveness has made it fit to be implemented in a wide array of software and hardware[1] related to multimedia applications.

However, its generic design made to cover a wide array of multimedia fronts, lack of acknowledgement or handshaking an one-way communication mandates a less abstract and more specific protocol that, in principle, could be constructed over OSC. The Virage project [AAl+10] has designed a multimedia protocol on top of OSC called *Minuit*.

The *Minuit* protocol implements the concept of *name-spaces* with a tree structure of nodes, each node has a set of attributes and is aware of the types and nodes under it. The client is capable of *discovering* the attributes and children of a certain node, *getting* and *setting* the value of a node and active *listening* the changes in value of a node.

Assume a name-space is the *OSC Address Space* of an *OSC Server* and a certain node is an *OSC Address*, let A be a server and B a client:

Discovering:

$$B \rightarrow ?namespace/Address$$
$$A \rightarrow : namespace/Address\ nodes = \{node1, node2, ...\}\ types = \{type1, type2, ...\}$$
$$attribitues = \{attr1, attr2, ...\}$$

Getting:

$$B \rightarrow ?get/Address : attribute$$
$$A \rightarrow : get/Address : attribute\ value$$

Setting:

$$B \rightarrow /Address \qquad \text{(Standard OSC Message)}$$
$$A \rightarrow No\ Reply$$

[1] For a full list, the reader might want to see http://opensoundcontrol.org/implementations

Listening:

$B \rightarrow ?listen/Address : attribute$ (Enable or disable listening of value)
$A \rightarrow : listen/Address : attribute$ (Will send value every time it changes)

Basically, a *discover* signal can be sent to a node(*OSC Address*) in a namespace(*OSC Address Space*), this will return a detailed description of the node including children, their types and attributes. A *get* message sent to a node will be replayed by the actual value of the attribute being asked and a *set* message is a standard *OSC Message* to a node setting its attributes. The user may also send a *listen* order to a node where in this case the server will send a replay to the client each time the node is updated.

A node is either of type `data`(when its a leaf) or of type `container` when it has children. The attributes a node possesses are:

- **value:** only on leaves
- **type:** may be:

  `integer, decimal, string, boolean, enum, array, anything`
- **priority:** 0 means none
- **range:** two values indicating the range of the node's value
- **comment:** text commentary.

An extended illustrative example of the usage of the *Minuit* protocol is in appendix A.2.

2.4 Property verification in Timed Systems

An important feature of any model of computation is the ability to perform verification over them. As it is the case in this work, the concept of time could also be present in the model whereas its time aspects play a major role in the description of any system constructed with it.

Verification of properties or specifications in timed computational models has been tackled before with a plethora of formalism and structures. [Aks+12] summarizes a large corpus of techniques focusing on verification of robustness requirements, also results of decidability, complexity and implementations are also summarized in a systematic order.

A common and famous abstract structure with a native definition of time are the *timed automata*, much work has been devoted to provide this structure

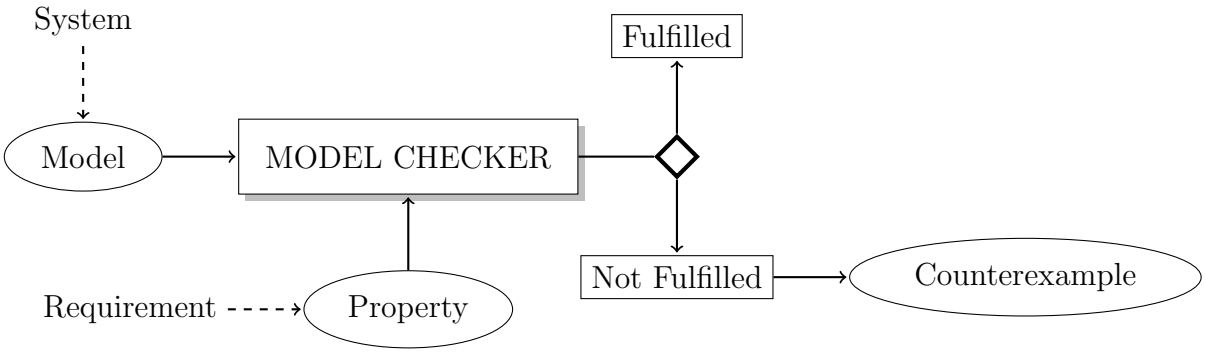


Figure 2.8: Model Checking scheme

with reasoning techniques. In [BCH12] an effort is made to embody networks of timed automata with *time traces*, a formalism to describe real-time distributed systems.

Of special importance is the work in [Alp+06], in here `tccp` programs are checked against a property expressed as a logical LTL formula(see section 2.4.1). The main idea steams from translating the `tccp` program onto an augmented LTL logic(with explicit notion of time) and verifying the property afterwards.

The problem with *deductive* methods as described above are that most of the work on proofs have to be done by hand or aided by computational theorem proving. For certain kind of timed systems the verification can be carried out automatically in its entirety by a approach called *Model Checking*[Lyg04].

Model Checking basically is a black box program where the system encoded in a programming language and the property specification expressed in an appropriate language or formalism are given to the computer to verify. After some minutes/hours/days/weeks the computer either crashes due to insufficient memory or returns with a yes/no answer and a counter example[Lyg04]. Figure 2.8 explains the Model Checking data flow.

A review of model checking techniques for concurrent languages is in [ORV13] including kripke structure-based techniques to mitigate the *state explosion* problem inherent to *Model Checking*. [Ari12] develops a kripke structure-based model checker for the `tcc` calculus, this is the base reference on which this work extends to `NTCC` (see chapter 4 for details). A model checker for `NTCC` is in [Gro13], the main drawback of this work is the *state explosion* problem which is alleviated by bounding the time interval of the check.

2.4.1 CLTL

In the last section it was clarified that *Model Checking* required a specification of a property for the computer to analyze and check. This property can be naturally expressed as a logic formula but a great concern lies within what type of logic is indeed the most appropriate to rationalize about the system specification.

In [Pnu77] temporal logics were introduced to the computer science scene, they provide a logical form of reasoning about time in a discrete fashion. Extensions and variations on this logic have been proposed, and of special relevance is the LTL(Linear Time Logic) where a formula is constructed with the following operators and syntax[HR04]:

$$\phi := \texttt{true} \mid \texttt{false} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \circ\,\phi \mid \phi_1 \mathcal{U} \phi_2$$

`true`, `false` are the basic truth values, p is an atomic proposition and $\neg$ and $\wedge$ have the same meaning from propositional logic. The operator $\circ\phi$ means that ϕ will hold `true` one **time unit** later and the operator $\phi_1 \mathcal{U} \phi_2$ states that ϕ_1 holds `true` **until** ϕ_2 is `true`.

CLTL, proposed in [Val05], is an extension of LTL in order to deal with `NTCC` programs. Its atomic propositions are constraint formulae as explained in section 2.1.1, figure 2.9 shows the **CLTL** syntax. Operators are dotted in order to differentiate them from their first-order logic counterparts.

As said before, operator 2.18 states that constraints are the atomic propositions, operators 2.19 and 2.19 are the basic truth values. Operators 2.22, 2.23 and 2.24 behave similarly from propositional logic, the conjunction is defined as $\varphi_1 \dot{\vee} \varphi_2 \stackrel{\text{def}}{=} \dot{\neg}\,(\dot{\neg}\varphi_1 \dot{\wedge} \dot{\neg}\varphi_2)$ and implication is defined as $\varphi_2 \dot{\rightarrow} \varphi_2 \stackrel{\text{def}}{=} \dot{\neg}\varphi_1 \dot{\vee} \varphi_2$.

Operator 2.25 hides variable x in the formula φ, operator 2.26 is similar to its LTL counterpart. Operator 2.27 is defined as $\Diamond\varphi \stackrel{\text{def}}{=} \dot{\texttt{true}}\,\mathcal{U}\varphi$ and intuitively means that the formula φ will eventually hold. Operator 2.27 is defined as $\Box\varphi \stackrel{\text{def}}{=} \dot{\neg}\Diamond\dot{\neg}\varphi$ and intuitively means that the formula φ always eventually hold.

As an example, suppose the binary state *switch* is a first-floor elevator request button, to ensure that the request is fulfilled one would write:

$$\varphi_{\text{example}} = \Box\,(switch = \texttt{on} \dot{\rightarrow}\ \circ\,(\Diamond\, floor = 1 \wedge switch = \texttt{off}))$$

$$
\begin{aligned}
\varphi \quad &:= c && (2.18)\\
&\mid \quad \dot{\texttt{true}} && (2.19)\\
&\mid \quad \dot{\texttt{false}} && (2.20)\\
&\mid \quad \dot{\neg}\varphi && (2.21)\\
&\mid \quad \varphi_1 \dot{\wedge} \varphi_2 && (2.22)\\
&\mid \quad \varphi_1 \dot{\vee} \varphi_2 && (2.23)\\
&\mid \quad \varphi_1 \dot{\rightarrow} \varphi_2 && (2.24)\\
&\mid \quad \dot{\exists}_x \varphi && (2.25)\\
&\mid \quad \circ\, \varphi && (2.26)\\
&\mid \quad \Diamond \varphi && (2.27)\\
&\mid \quad \Box \varphi && (2.28)
\end{aligned}
$$

Figure 2.9: **CLTL** Syntax

Closure of a Formula

The closure of a formula is defined as the set of all sub-formulas whose truth value can influence in turn the truth value of the whole formula. The set $CL(\varphi)$ is constructed as[Ari12]:

- $\varphi \in CL(\varphi)$
- $\dot{\neg}\phi \in CL(\varphi)$ iff $\phi \in CL(\varphi)$
- if $\phi_1 \dot{\wedge} \phi_2$ or $\phi_1 \dot{\vee} \phi_2$ or $\phi_1 \dot{\rightarrow} \phi_2 \in CL(\varphi)$ then ϕ_1 and $\phi_2 \in CL(\varphi)$
- if $\dot{\exists}_x \phi \in CL(\varphi) \in CL(\varphi)$ then $\phi \in CL(\varphi)$
- if $\circ\, \phi \in CL(\varphi)$ then $\phi \in CL(\varphi)$
- if $\dot{\neg} \circ \phi \in CL(\varphi)$ then $\circ \dot{\neg} \phi \in CL(\varphi)$
- if $\Box\phi \in CL(\varphi)$ then ϕ and $\circ\Box\phi \in CL(\varphi)$
- if $\Diamond\phi \in CL(\varphi)$ then ϕ and $\circ\Diamond\phi \in CL(\varphi)$

To understand the concept of *influencing* the truth value it is important to, at the very least, look at one of these closure construction rules. The truth value of formula $\Box\phi$ directly depends the truth value of its sub-formula ϕ, but in spite of this it **also** depends whether ϕ is always true, therefore the truth value of the initial formula in the next time unit($\circ\Box\phi$) has to be in its closure.

Continuing the elevator example, the closure of φ_{example} is:

$$
\begin{aligned}
CL(\varphi_{\text{example}}) = \{ & \Box\left(switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right), \\
& \dot{\neg}\left(\Box\left(switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right)\right), \\
& \circ\,\Box\left(switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right), \\
& \dot{\neg}\left(\circ\,\Box\left(switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right)\right), \\
& \circ\,\dot{\neg}\left(\Box\left(switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right)\right), \\
& switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right), \\
& \neg\left(switch = \texttt{on} \dot{\rightarrow} \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right), \\
& switch = \texttt{on}, \neg\left(switch = \texttt{on}\right), \circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right), \\
& \dot{\neg}\left(\circ\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right)\right), \\
& \circ\,\dot{\neg}\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right), \\
& \Diamond\, floor = 1 \wedge switch = \texttt{off}, \dot{\neg}\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right), \\
& \circ\,\Diamond\, floor = 1 \wedge switch = \texttt{off}, \dot{\neg}\left(\circ\,\Diamond\, floor = 1 \wedge switch = \texttt{off}\right), \\
& \circ\,\dot{\neg}\left(\Diamond\, floor = 1 \wedge switch = \texttt{off}\right), \\
& floor = 1 \wedge switch = \texttt{off}, \dot{\neg}\left(floor = 1 \wedge switch = \texttt{off}\right)\}
\end{aligned}
$$

Chapter 3

Project React+ and Problem Overview

The work presented herein is part of the project REACT+ and the author of this document is ascribed to the research group AVISPA(Ambientes Visuales de Programación Aplicativa). This chapter states in a concise manner the problem targeted within this work; section 3.1 gives details on the preceding and actual projects of which this work is part of and section 3.2 formally describes the problem and defines its goals, motivation and scope.

3.1 Previous Work

3.1.1 REACT

Project REACT(Robust Theories for Emerging Applications in Concurrency Theory)[AVI06] was a joint effort between the AVISPA research group, the IRCAM institute and the INRIA Team Comète. The motivation was to model through concurrency theory emerging applications in certain areas, namely; security protocols, biological systems and multimedia semantic interaction.

A central approach point in this project was the **reachability analysis** as means to provide for formal reasoning in any application on the fields mentioned above. The strategy was to dispense the process calculus NTCC (see section 2.1.1) with a reachability robust theory and tailor it to each of these fields.

Some of the results of REACT were; in [AGP06] a plethora of biological systems are modelled in NTCC with conclusions of its expressiveness in this type of applications, in [Gut+07] an approach is presented to correctly model and verify

biological systems, in [Her+11] using the partial information methodology of NTCC a model of cellular communication system is constructed. The work presented in [OR09] extends NTCC to model dynamic multimedia interactions and in [OV08] the same extension is used to present models of applications in networking security. Finally an extension of NTCC to provide probability operators is presented in [PR08b].

3.1.2 REACT+

Garavel in [Gar08] pointed that formal method of computing were no longer a pen-and-paper projects for mathematicians. If a formal language was not equipped with proper programming environments, software and tools, adoption by the industry would be very unlikely.

The project REACT+[AVI12] was born to compensate this issue with respect to NTCC and is set to develop the underlying theory and provide automatic verification and simulation techniques. This is tailored to allow an integrated environment for modelling, executing and verifying applications.

Automatic verification of reactive systems poses a fundamental challenge due to the state explosion problem[AVI12]. The approach taken in REACT+ is to identify parts of the programs encoded in NTCC that are subject to automatic verification and develop techniques to machine-assist the verification of a property. To reduce the state space, it is proposed to use *abstract interpretation and types techniques*[CC92] which allow to obtain a compact representation suitable for processing.

Of supreme importance to the objective of this work is the specific section addressing multimedia interaction:

> *Complex multimedia systems are usually built out of third-party components suited to specific domains, such as sound processing. We plan to use techniques such as abstract interpretation to define rigorously the interaction of ntcc models with such components. We will use our techniques and tools to verify coherence properties of composite interactive multimedia systems with sound processing [applications]*

3.2 Problem Statement

3.2.1 Definition

This document presents the work directed to solve the goal of *multimedia interaction* within the REACT+ project, the problem is formally defined as:

> *The definition, implementation and integration of multimedia interaction into the NTCC framework for simulation and verification purposes.*

The effort will focus on integrating and communicating a novel score follower called ANTESCOFO(see section 2.2.1) with NTCC through a multimedia protocol and will also center around techniques of model checking for property verification. The selection of ANTESCOFO as the multimedia application is due to its novelty and its origin from IRCAM, a collaborator institute in the REACT+ project.

3.2.2 Application

As REACT+ seeks to develop a programming framework for NTCC , the need arises to correctly integrate applications from different natures, such as multimedia, to this NTCC framework. Therefore, consistent intercommunication and feedback is mandatory for more specialized third-party applications to enjoy the formal reasoning power NTCC has to offer.

For example, if an application provides the capability for rhythm recognition and classification, it would be desirable after identifying the actual rhythm of a signal if another application was able to check patterns of dance rhythms such as a *saltarello*, this would provide the user with automatic classification of songs.

From the research point of view, computer music has also leaned towards a consistent base of abstract mathematical structures to represent musical events with the purpose of isolating, decomposing and manipulating such events . Moreover, this work aims not just to be part of REACT+ goals and scope but also aims to contribute to the field of computer music by providing ways of formal verification of musical properties in complex music settings like orchestation and improvisation.

3.2.3 Goals

This section lists the main and specific goals of the work presented here in an unambiguously manner, they are as follows.

Main

To communicate the automatic score follower system ANTESCOFO with the NTCC framework of the project REACT+ and also integrate its action language for interaction and verification of system properties.

Specific

1. Analyze ANTESCOFO's output data and communicate it with the Oz language through a multimedia protocol.
2. Develop or reuse a multimedia protocol for communication(input and output channel) and synchronization(on the sense of reactive systems) with the NTCC framework using Oz.
3. Integrate ANTESCOFO's internal action language in the framework(using either Oz or a communication interface) for real-time use.
4. Test properties(rhythmical, tonal or improvisational) and/or synchronize NTCC actions in actual musical pieces using the framework.

3.2.4 Scope

The scope of the work herein falls alongside the expected results of REACT+, they are presented here in a more detailed way.

- An implementation of a multimedia protocol for the Oz language.
- Efficient communication of a real multimedia application with the framework.
- Correct synchronization of actions between a multimedia application and the NTCC framework.
- Automatic verification of musical properties in at lest one known musical work(i.e. Piano Phase by Steve Reich).

Chapter 4

Musical Property Verification

The following chapter depicts in precise detail the complete scheme devised to have automatic verification of musical properties. The solution proposed in here is made up of several interconnecting modules, therefore a global explanation precedes an individual explanation of each one of these modules.

Section 4.1 details the overall design and data flow of the solution and also explains the input and output of data from one module to the other. Section 4.2 particularizes in a terse mode the development of each module that did not have an appropriate previous implementation for usage in this project.

4.1 Design

The proposed solution has a general method consisting of the following steps:

1. Create an ANTESCOFO score which interactively constructs the corresponding `NTCC` process(the system).
2. Construct the **CLTL** formula which represents the property to be verified(the specification).
3. Start the OSC listening client.
4. Either play the score on-site or feed an acoustic signal to ANTESCOFO in order to interactively construct the `NTCC` process.
5. Analyze the output of the Model Checker module.

This solution design is usefull in the sense of analysing or verifying precise musical aspects of the relation between some score and a particular performance from it. For example, suppose a composer wants to enforce a certain temporal

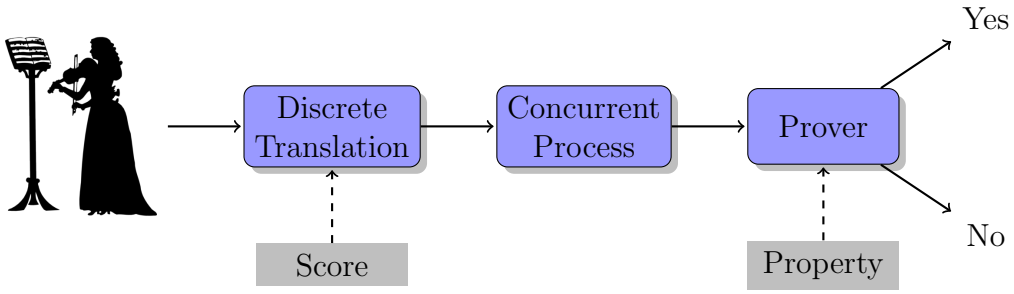


Figure 4.1: Overall design scheme of automatic musical property verification

characteristic from a handful of automatic generated recordings of a musical piece he crated. If not long he may do it by hear or let the method decide automatically whether this is true or not, he may even thought out other properties just for the sake of feedback into the construction process of his work.

More so, it is not obligatory to have a performance of a certain piece, the user may wish to only check abstract properties of the score at hand before he decides to ultimately perform it or have it performed. ANTESCOFO admits sequencing of scores without input audio signal, something similar to have a perfect performance of a score.

Figure 4.1 plots the data flow of the general solution, the *discrete translation* module is done by ANTESCOFO(see section 2.2.1) with a *score* being fed offline. The score is then translated to an `NTCC` process in the module *concurrent process*(see section 4.1.2). This process is interactively passed to the *prover* module(see section 4.2.2), the property to be verified is given priory. At the end of the construction of the `NTCC` process an answer is the final output.

4.1.1 Communication Strategy

A critical transport of information in figure 4.1 is in-between the *discrete translation* step and the *concurrent process* module. This is because data is exchanged between two different programs(Pure Data and Oz) and also this exchange needs to be multimedia oriented and high level. The `OSC` specification(see section 2.3) is an appropriate candidate for this requirement and the *Minuit* protocol, built on top of `OSC`, makes a reasonable choice.

Section 4.2.1 gives insight into the implementation of this protocol on the Oz language.

4.1.2 Musical translation to NTCC

As mentioned before, the *concurrent process* module is where the translation of the discrete musical event to the corresponding NTCC process takes place. A direct translation of any musical event to a computational agent is not trivial, mainly because the interpretation of the event in terms of any computational goal differs broadly.

A possible formalized translation with a consistent set of logical rules could be developed but exceeds the scope of this project. A more centralized form would be to associate a constraint to each event recognizable in the *discrete translation* module, but this results in an inflexible output for the program to verify.

In spite of this, the solution developed leaves this translation to be specified by the user inside the ANTESCOFO score. Therefore each musical event recognized may trigger an action that interactively translates to a NTCC program. This solution is more flexible but demands careful attention and knowledge from the user to correctly assemble the NTCC translation. Section 5.2.2 further comments on this issue.

The possible actions that can be triggered by an event in the ANTESCOFO score are as follows:

- `/ntcc min max var1 var2 ...`: Initiates the NTCC process and a store with variables `var1`, `var2`, etc. The possible values of these variables are in the integer interval [`min` − `max`].
- `/ntcc/tell var rel value`: Poses a constraint into the store. The possible relations are `=:`, `>:`, `<:`, `>=:`, `=<:` and `\\=:`.
- `/ntcc/when var rel value`: Conditions the proceeding NTCC agent by the specified constraint.
- `/ntcc/unless var rel value`: Conditions the proceeding NTCC agent by the absence of the specified constraint.
- `/ntcc/par`: Constructs a parallel execution for the proceeding NTCC agents. The scope of this parallel operator is closed by another occurrence of `/ntcc/par`.
- `/ntcc/sum`: Constructs a non-deterministic choice for the proceeding NTCC agents. The scope of this parallel operator is closed by another occurrence of `/ntcc/sum`.
- `/ntcc/next units`: Delays the proceeding NTCC agent by the specified number of time units.
- `/ntcc/rep`: Replicates the proceeding NTCC agent.
- `/ntcc/star`: Non-determinedly delays the defined NTCC agent.

The frame below is an example of an ANTESCOFO score of a full ascending C major scale(in midi values), the constructed NTCC process is:

$$!\,((\textbf{when } x = 60 \textbf{ next tell } x = 67) \parallel (\textbf{when } x > 67 \textbf{ next tell } x = 72))$$

```
BPM 60

EVENT 0.0
      osc msg /ntcc 0 100 x
      osc msg /ntcc/rep
      osc msg /ntcc/par

NOTE 60 1.0 Tonic
     osc msg /ntcc/when x =: 60

NOTE 62 1.0 Supertonic

NOTE 64 1.0 Mediant

NOTE 65 1.0 Subdominant

NOTE 67 1.0 Dominant
     GFWD 0.0 Group4
     {
        0.0 osc msg /ntcc/next 1
        0.0 osc msg /ntcc/tell x =: 67
     }

NOTE 69 1.0 Submediant
     osc msg /ntcc/when x >: 67

NOTE 71 1.0 Subtonic

NOTE 72 1.0 Octave
     GFWD 0.0 Group7
     {
        0.0 osc msg /ntcc/next 1
        0.0 osc msg /ntcc/tell x =: 72
        0.0 osc msg /ntcc/par
     }
```

4.1.3 Property Checking Technique

For proving properties of an NTCC process, the *prover* module implements a model checker. It is based on the tcc model checker implemented in [Ari12] with a partial extension to NTCC done in [Tor13] and ported to Oz. To couple with the interactive nature of this project, the model checker has also been extended to deal with incremental construction of **CLTL** formulas and NTCC processes[1].

[1]For the full source code and examples of this module the reader may refer to http://cic.javerianacali.edu.co/~ysperchy/ntccModelChecker.zip

To deal with the inherent state explosion problem of model checking, the model structure to represent NTCC systems are kripke structures devoid of any internal transitions. Section 4.2.2 illustrates the ideas behind this model checker as well as the abstract concepts used in it.

4.2 Implementation

4.2.1 OSC and *Minuit* Protocol

In spite of the growing popularity and support of the OSC protocol, an official implementation in the Mozart/Oz language is still non-existent, a base C implementation of OSC is supposed to be a starting point for any future coding. A full list of the software and hardware implementations of OSC is in [CNM12].

The audio programming language *Strasheela* has a Mozart implementation of OSC in [And12]. At a closer analysis this implementation is found to be unsuited to this work mainly for two reasons: it depends on the UNIX implementations of the sendOSC and dumpOSC programs which in turn makes it not multi-platform, also it depends on various libraries specifically coded for the *Strasheela project*.

Moreover, using sendOSC and dumpOSC makes the implementation less efficient(a claim by the author himself) which would render it improper because the project REACT+ aims for the speed benefits of concurrency computation. Likewise, to pass the data onto Mozart this *strasheela* implementation relies on netcat(another UNIX program), and dumpOSC has to be run necessarily in a console(xterm). Finally, the version of Netcat included in MacOS versions does not work correctly in UDP mode.

Thus, this work provides an Oz implementation of OSC which is necessarily **thread-safe** due to the concurrent nature of NTCC . Mozart/Oz has an interface for *Bite Strings* but it does not offer byte handling capabilities for any byte ordering demand, because of this an OSC implementation directly in Oz is not possible instead recurring to the C interface.

The implementation provided here, called OSC-Oz, is based on *OSCPack*, a C++ OSC implementation made in [Ben10]. Appendix B.1 details the complete specification of this API for usage in the Oz language[2].

Figure 4.2 shows the data-flow for the OSC-Oz module on both packing and unpacking scenarios of OSC Packets. For packing a message the user must first begin a packet and optionally begin a bundle, then start a message and add

[2]For the full source code and examples of this library the reader may refer to http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:osc-oz

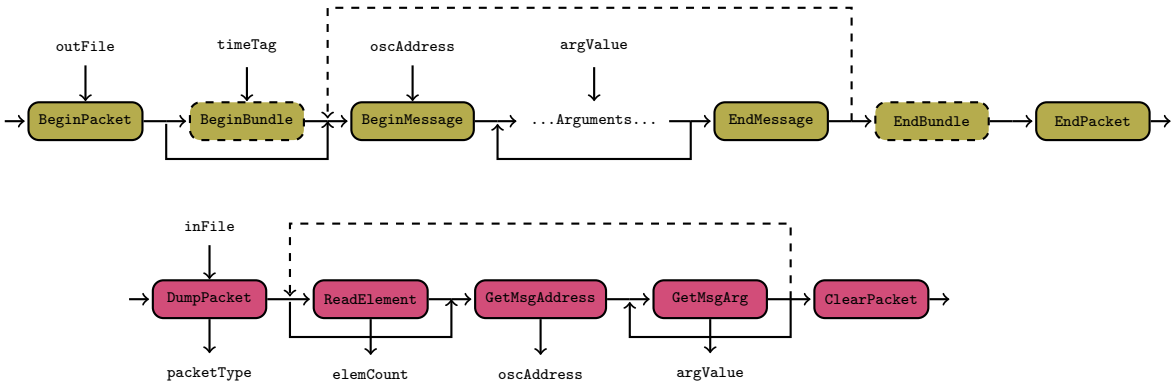


Figure 4.2: OSC-Oz data-flow

arguments as needed before closing such message, the process of the message creation repeats as desired if a bundle was begun. Finally the user closes the packet and it is left(by means of a file) for its usage in Mozart.

The process of unpacking is similar; the user dumps the received packet(also, by means of file) and starts to read each element contained in it in case of a bundle. After this, the address and arguments of each message are available for query before the user clears the packet from the primary memory. The **thread-safe** requirement is assured by assigning a unique ID number for each processed packet either for packing or unpacking operations.

The OSC-based *Minuit* protocol[Vir09b] makes a clear definition of data hierarchy and client/server distinction. A set of implementations of this protocol is in [Vir09a] but unfortunately, there does not exists a Mozart implementation end neither of all the implementations presented there are for the version 0.3.

This work also provides a Mozart/Oz implementation of the *Minuit* protocol which is **thread safe** and offers client and server functionality. Appendix B.2 fully documents the API of this Oz module.

As the ANTESCOFO implementation provided for the project is native to Pure Data, a functional *Minuit* implementation on this same language has to be present too. The official implementation provided by the Virage project is not up-to-date, therefore an implementation for Pure Data of the client-side functionality is also offered in this work. Appendix B.2.1 details the Pure Data patch made for this goal.

Because the *Minuit* protocol uses non-standard OSC messages for its queries, a slight modification to the OSC implementation for Pure Data in [Pea11] has to be done in order to work.

4.2.2 Model Checking

The model checking library for `NTCC` was directly coded in Mozart/Oz using its built-in constraint engine, it is an integral part of the `REACT+` project goals. Currently it allows for a step-by-step creation of the system model(the `NTCC` process) and the system specification(the **CLTL** formula).

It has 4 modules implemented; (1) a `ltlFormula` interface, (2) a `ntccProcess` interface, (3) a `ntccStructure` interface for handling the kripke structures from and `NTCC` process and (4) a `ntccModelChecker` interface for performing the model check . In appendix C.1 the interfaces of these modules are exhaustively documented.

The main algorithm for the model checking procedure is in listing 1. To understand the algorithm better note that, up to line 3 the goal formula is devoid of any $\dot{\rightarrow}, \dot{\vee}$ and $\Box$ operators through the following equivalences; (a) $\varphi_1 \dot{\rightarrow} \varphi_2 \equiv \dot{\neg}\varphi_1 \dot{\vee} \varphi_2$, (b) $\varphi_1 \dot{\vee} \varphi_2 \equiv \dot{\neg}(\dot{\neg}\varphi_1 \dot{\wedge} \dot{\neg}\varphi_2)$ and (c) $\Box\varphi \equiv \dot{\neg}(\Diamond \dot{\neg}\varphi)$.

Algorithm 1 Main Model Checking Algorithm

Require: *ntccProcess* and *ltlFormula* are properly constructed

```
1:  function ModelCheck(ntccProcess, ltlFormula)
2:      negFormula ← NegateFormula(ltlFormula)
3:      formula ← TransformFormula(negFormula)        ▷ Remove →̇, ∨̇ and □
4:      closure ← FormulaClosure(formula)
5:      extendedStruct ← ExtendedStructure(ntccProcess)
6:      reducedStruct ← ReducedStructure(extendedStruct)
7:      MCGraph ← ModelCheckingGraph(reducedStruct, closure)
8:      result ← true
9:      for all graph ← SCCGraph(MCGraph) do                ▷ For each SCC
10:         if InitialEntailment(graph, formula) then
11:             if IsSelfFulfilling(graph) then          ▷ Is the SCC self-fulfilling?
12:                 result ← false
13:             end if
14:         end if
15:     end for
16:     return result
17: end function
```

This leaves the formula only with operators $\dot{\neg}, \dot{\wedge}, \circ$ and $\Diamond$. This is of great help because stores can be constructed by a series of constraint conjunctions($\dot{\wedge}$) and nodes in graphs representing the `NTCC` process can be coherently connected by $\circ$ operators. The operator $\Diamond$ can be checked with a concept called *promised formula*[Ari12] that will be explained shortly.

Continuing the explanation, the algorithm calculates the closure of the goal formula(line 4), calculates the extended kripke structure of the NTCC specification and the reduces it(lines 5 and 6) and constructs the model checking graph(line 7) from these calculations. The search of whether the formula is fulfilled or not by the specification takes the reminder of the algorithm(from line 9 to line 15).

Extended and Reduced Kripke Structure

The general approach when implementing model checking is to transform the system model to an abstract structure containing the relevant aspects of the model and more kin to navigate and search. The implementation in here uses a kripke structure(a non-deterministic automaton) $M = \langle S, I, T, R \rangle$ where S is a set of states, $I \subseteq S$ are the initial states, $T = \{i, t\}$ are the types of transitions, i means an internal transition and t means a temporal transition, and $R \subseteq S \times S \times T$ are the transitions of the automaton from one state to another.

Each state $S_i = \langle St, Li, Lt \rangle$ contains a store $St \subset 2^c$(which is a conjunction of constraints) and labels of internal L_i and temporal L_t NTCC agents scheduled to execute. The kripke structure is constructed off the NTCC process with the next set of rules:

- P_1 (Initial Process):
  1. Add the initial state $S = \langle St, Li, Lt \rangle$
  2. $St = \emptyset$
  3. $Li = P_1$
  4. $Lt = \emptyset$
- $\texttt{tell}_n\ c$:
  1. Add a state $S' = \langle St', Li', Lt' \rangle$ with an internal transition from its parent state $S = \langle St, Li, Lt \rangle$
  2. $St' = St \wedge c$
  3. $Li' = Li - \{\texttt{tell}_n\}$
  4. $Lt' = Lt$
- $\texttt{when}_n\ c\ \texttt{do}\ P$:
  1. Add two states $S'_1 = \langle St'_1, Li'_1, Lt'_1 \rangle$ and $S'_2 = \langle St'_2, Li'_2, Lt'_2 \rangle$ with internal transitions from their parent state $S = \langle St, Li, Lt \rangle$
  2. $St'_1 = St \wedge c$, $St'_2 = St \wedge \neg c$
  3. $Li'_1 = Li - \{\texttt{when}_n\} \cup \{P\}$, $Li'_2 = Li - \{\texttt{when}_n\}$
  4. $Lt'_1 = Lt'_2 = Lt$
- $\texttt{unless}_n\ c\ \texttt{next}_{n+1}\ P$:
  1. Add two states $S'_1 = \langle St'_1, Li'_1, Lt'_1 \rangle$ and $S'_2 = \langle St'_2, Li'_2, Lt'_2 \rangle$ with internal transitions from their parent state $S = \langle St, Li, Lt \rangle$
  2. $St'_1 = St \wedge c$, $St'_2 = St \wedge \neg c$

3. $Li'_1 = Li - \{\texttt{unless}_n\}$, $Li'_2 = Li - \{\texttt{unless}_n\} \cup \{\texttt{next}_{n+1}\ P\}$
4. $Lt'_1 = Lt'_2 = Lt$

- $P_1 \parallel_n \ldots \parallel_n P_m$:
  1. Add a state $S' = \langle St', Li', Lt' \rangle$ with an internal transition from its parent state $S = \langle St, Li, Lt \rangle$
  2. $St' = St$
  3. $Li' = Li - \{\|_n\} \cup \{P_1 + \ldots + P_m\}$
  4. $Lt' = Lt$
- $P_1 +_n \ldots +_n P_m$:
  1. Add m states $S'_i = \langle St'_1, Li'_1, Lt'_1 \rangle$ where $1 \leq i \leq n$ with internal transitions from their parent state $S = \langle St, Li, Lt \rangle$
  2. $St'_i = St$
  3. $Li'_i = Li - \{+_n\} \cup \{P_i\}$
  4. $Lt_i = Lt$
- $\texttt{next}_n\ P$
  1. Add a state $S' = \langle St', Li', Lt' \rangle$ with an internal transition from its parent state $S = \langle St, Li, Lt \rangle$
  2. $St' = St$
  3. $Li' = Li - \{\texttt{next}_n\}$
  4. $Lt' = Lt \cup \{P\}$
- $!_n P$
  1. Add a state $S' = \langle St', Li', Lt' \rangle$ with an internal transition from its parent state $S = \langle St, Li, Lt \rangle$
  2. $St' = St$
  3. $Li' = Li - \{!_n\} \cup \{P\}$
  4. $Lt' = Lt \cup \{!_n P\}$
- $\star_n P$
  1. Add two states $S'_1 = \langle St'_1, Li'_1, Lt'_1 \rangle$ and $S'_2 = \langle St'_2, Li'_2, Lt'_2 \rangle$ with internal transitions from their parent state $S = \langle St, Li, Lt \rangle$
  2. $St'_1 = St'_2 = St$
  3. $Li'_1 = Li - \{\star_n\} \cup \{P\}$, $Li'_2 = Li - \{\star_n\}$
  4. $Lt'_1 = Lt$, $Lt'_2 = Lt \cup \{\star_n\}$
- $S = \langle St, \emptyset, Lt \rangle$:
  1. Add a state $S' = \langle St', Li', Lt' \rangle$ with a temporal transition from the state S
  2. $St' = \emptyset$
  3. $Li' = Lt$
  4. $Lt' = \emptyset$

The first rule sets the initial state where its internal labels are the main NTCC process, the store and the temporal labels are empty. The last rule expands states where there is no internal process unrealized, this ensures the whole structure is a full representation of all the possible paths the process may lead to. To keep

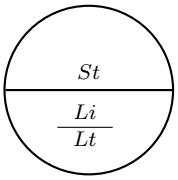


Figure 4.3: NTCC state the kripke structure

the structure finite, every time a state S' is added if another present state S is the same the transitions is made to this state S instead of creating a new state S'.

Figure 4.4 shows the kripke structured formed by the labeled NTCC process:

$$\mathbf{when}_1\ x > 0\ \mathbf{do}\ \mathbf{next}_2\ (\star_3\ (!_4\ \mathbf{tell}_5\ x = 0))$$

Single tipped arrows($\rightarrow$) mean internal transitions, double tipped arrows($\twoheadrightarrow$) mean temporal transitions and double circles are initial states. Figure 4.3 details the NTCC state information on each node.

A significant reduction on the number of states of the structure can be achieved by the concept of *unobservable* transition from [NV04]. This idea is originally applied to tcc but can be carried along to NTCC where the same concept of *quiescent point* in a time unit is kept. The main idea is to remove the internal transitions as they are regarded *unobservable* and only care about temporal or *observable* transitions. There are two rules to perform this reduction:

1. If there is a sequence of internal transitions between two temporal transitions, then delete transitions except the last one(figure 4.5a).
2. If there is a temporal transition leading to a state where there is a branching of internal transitions(such as the case of `when`) then clone the branching node as many times as there are internal branching transitions and set temporal transitions from the initial node to these cloned nodes(figure 4.5b).

Figure 4.6 shows the kripke structure obtained from reducing the structure in figure 4.4.

Model Checking Graph

To generate the model checking graph the kripke structure needs to be combined with the closure of the formula to be proven. This combination creates another graph where each state S_i of the kripke structure is associated to a subset C_i of the closure to create a new state $M_i = \langle S_i, C_i \rangle$ in the model checking graph. The

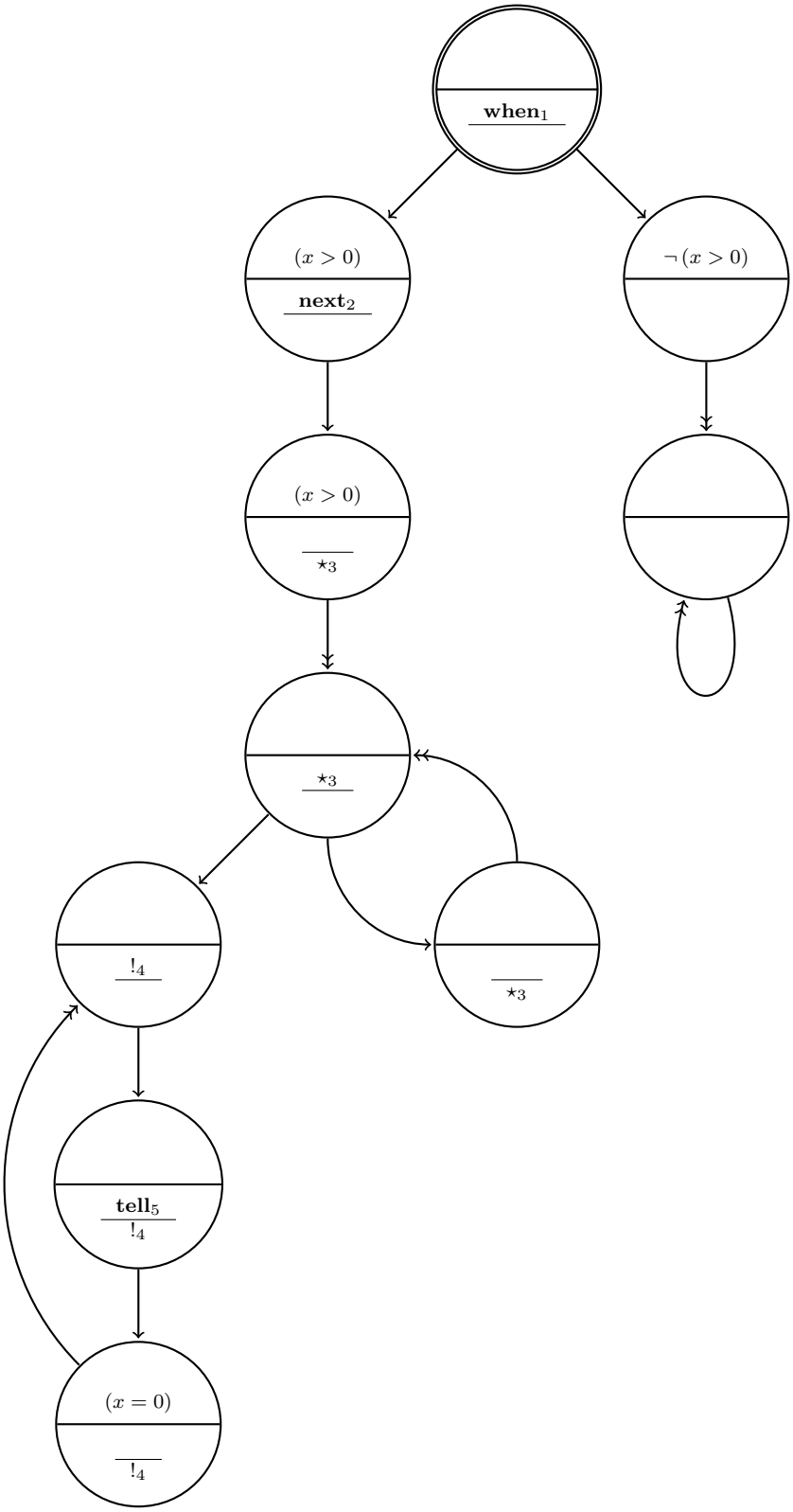
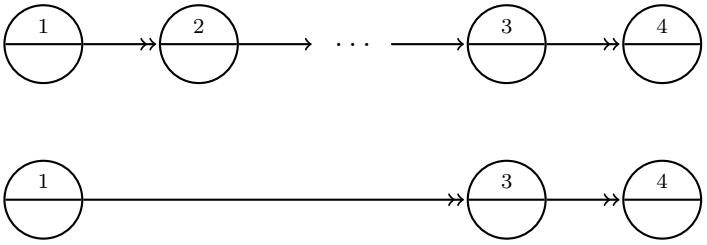
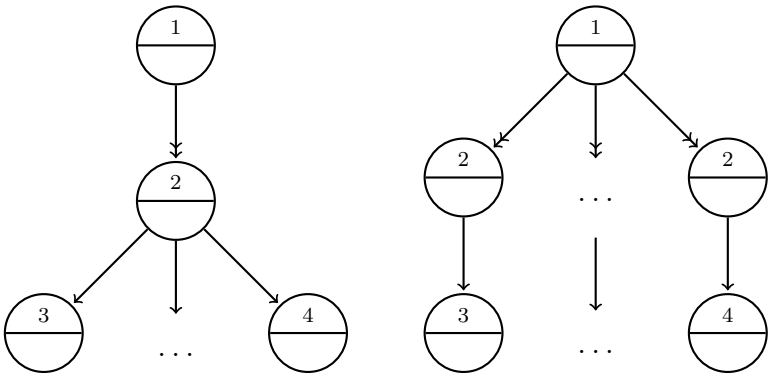


Figure 4.4: Krikpe structure for the NTCC process



(a) Internal sequence reduction



(b) Internal branching reduction

Figure 4.5: Unobservable state reductions

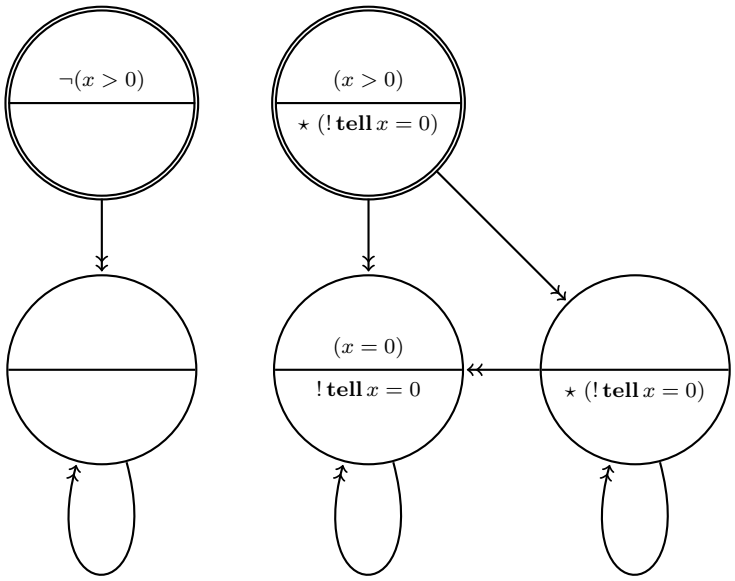


Figure 4.6: Reduced Kripke structure for the `NTCC` process

association is done in order to create a graph with the basic formulas from the closure that are consistent with that particular state.

Let the property formula be φ, $CL(\varphi)$ its closure, S_i any state from the kripke structure and $S_i(St)$ its store. The association of state M_i is done according with the next set of rules[Ari12]:

- for each atomic **CLTL** proposition p, $p \in C_i$ iff $p \in S_i(St)$
- for each $\phi \in CL(\varphi)$, $\phi \in C_i$ iff $\dot{\neg}\phi \notin S_i(St)$
- for each $\dot{\exists}_x\phi \in CL(\varphi)$, $\dot{\exists}_x\phi \in C_i$ iff $\dot{\exists}_x \in S_i(St)$
- for each $\phi_1 \dot{\wedge} \phi_2 \in CL(\varphi)$, $\phi_1 \dot{\wedge} \phi_2 \in C_i$ iff $\phi_1 \in S_i(St)$ and $\phi_2 \in S_i(St)$
- for each $\phi_1 \dot{\vee} \phi_2 \in CL(\varphi)$, $\phi_1 \dot{\vee} \phi_2 \in C_i$ iff $\phi_1 \in S_i(St)$ or $\phi_2 \in S_i(St)$
- for each $\phi_1 \dot{\rightarrow} \phi_2 \in CL(\varphi)$, $\phi_1 \dot{\rightarrow} \phi_2 \in C_i$ iff $\dot{\neg}\phi_1 \in S_i(St)$ or $\phi_2 \in S_i(St)$
- for each $\dot{\neg} \circ \phi \in CL(\varphi)$, $\dot{\neg} \circ \phi \in C_i$ iff $\circ \dot{\neg} \phi \in S_i(St)$
- for each $\Box\phi \in CL(\varphi)$, $\Box\phi \in C_i$ iff $\phi \in S_i(St)$ and $\circ \Box\phi \in S_i(St)$
- for each $\Diamond\phi \in CL(\varphi)$, $\Diamond\phi \in C_i$ iff $\phi \in S_i(St)$ or $\circ \Diamond\phi \in S_i(St)$

An edge in the model checking graph from node $M_i = \langle S_i, C_i \rangle$ to node $M_{i+1} = \langle S_{i+1}, C_{i+1} \rangle$ exists if the following two conditions are met: (1) A transition exists in the original structure between states S_i and S_{i+1} and, (2) for every $\circ\phi \in C_i$ then $\phi \in C_{i+1}$.

4.2.3 Searching the property satisfaction

With the model checking graph constructed, the model checking algorithm can now deduce if the requirement formula φ can be satisfied by the model. The main objective here is show that if there is no path in the model checking graph that is always consistent with $\dot{\neg}\varphi$ then φ is satisfied by the model, otherwise it is not.

A formula f is satisfied by a path of the model checking graph if two conditions are met:

1. f is entailed by the all nodes across the path until,
2. the path reaches a Self-fulfilling Strongly Connected Component(SCC) of the graph.

A SCC component of any graph G is defined as a maximal subgraph $S \subseteq G$ such that for every two nodes $S_1, S_2 \in S$ there must exist a path from S_1 to S_2 without leaving the boundary of S. Intuitively speaking, the SCC components of

the model checking graph are the states where loops in the execution of the `NTCC` model may happen.

A Self-fulfilling SCC component of the model checking graph G is a subgraph $S \subseteq G$ such that S is an SCC component of G and for every node $M = \langle S_k, C \rangle \in S$ and for every $\Diamond\phi \in C$ there exist another node $M' = \langle S'_k, C' \rangle$ such that $\phi \in C'$. In other words, an SCC component is Self-fulfilling if all promises(in terms of $\Diamond\phi$) are delivered within the SCC component. This avoids `NTCC` execution loops that never comply its $\star$ processes.

As mentioned before, the specification formula is left in terms if $\dot{\neg}, , \dot{\wedge}, \circ$ and $\Diamond$. A node entailing a formula f means that its store, expressed as conjunctions of propositions and negations of proposition, is consistent with f. A sequence of nodes(a path), by construction of the model checking graph, is automatically consistent with its $\circ$(`next`) agents and finally the Self-fulfilling property takes care of the $\Diamond(\star)$ agents.

Chapter 5

Results and Conclusions

A scheme for proving musical properties was proposed and developed in the last chapter. The remaining chapter of this document is devoted to detail some tests and show their results as well as present the main contributions, conclusions and possible directions of work.

Section 5.1 introduces a musical piece from the academic repertoire and displays three musical properties passed through the proposed scheme. These three properties exemplify how the scheme works and each one is given a musical significance along with an explanation of the correspondent output.

Section 5.2 concludes the document with important remarks of the work and future directions respectively.

5.1 Tests and Results

As stated before, this section presents three examples of proving musical properties from a well defined ANTESCOFO score. The system specification in each test is an `NTCC` process where its construction is specified in the scores and the **CLTL** formula(the requirement) is thought off a priory and programmed in the Oz client program.

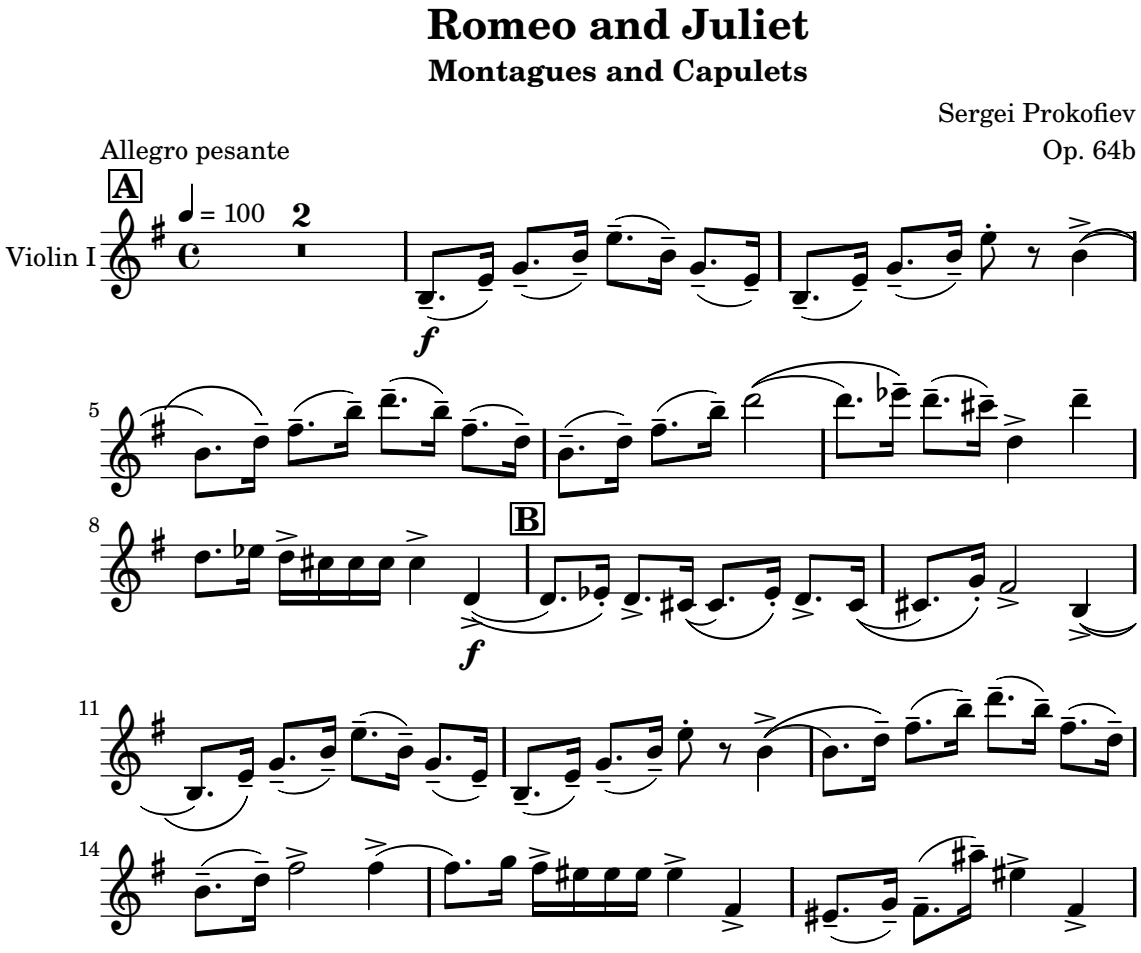
5.1.1 The Piece

All tests use the same score based on a single musical input, the music is *Dance of the Knights*[1](also known as *Montagues and Capulets*) from Russian composer

[1]Original Cyrillic title: Танец рыцарей

Sergei Prokofiev(⋆1891 - †1953). The piece itself was composed in the decade of 1930 as part of his ballet *Romeo and Juliet* Op.64 to great acclaim. Later the composer would adapt to piano and orchestra some music from this ballet in three suites(including Dance of the Knights, Suite No.2 Op.64b).

The examples presented use an excerpt of the first 16 bars from the first Violin, the score is depicted next and is transcribed from a scanned score from the IMSLP database [IMS11].



5.1.2 Musical Properties

First property

The first property relates to the melodic intervals marked in each bar of the first *period*[2]. They are the lowest and highest notes, figure 5.1 exemplifies this

[2]A period is a small self-contained musical idea, it is the fundamental piece for constructing larger forms in classical music.

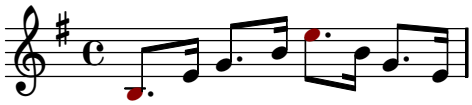


Figure 5.1: Interval notes of a bar in red color

concept.

The ANTESCOFO score in appendix D.1 dynamically creates the system specification reflecting this characteristic in a sequential fashion. As the musician plays the score, the `NTCC` process is created with the information from the score. Let $n = \langle p, d \rangle$ be a note of pitch p and duration d and $i = [n_l, n_h]$ an interval of lowest note n_l and highest note n_h. The created `NTCC` process is then:

$$\begin{aligned} SCORE_1 \stackrel{\text{def}}{=} & \textbf{tell}\ i = [\langle b, \text{♪.}\rangle, \langle e, \text{♪.}\rangle] \parallel \\ & \textbf{next}\ (\textbf{tell}\ i = [\langle b, \text{♪.}\rangle, \langle e, \text{♪}\rangle]) \parallel \\ & \textbf{next}^2\ (\textbf{tell}\ i = [\langle b', \text{♪.}\rangle, \langle d', \text{♪.}\rangle]) \parallel \\ & \textbf{next}^3\ (!\,\textbf{tell}\ i = [\langle b', \text{♪.}\rangle, \langle d', \text{𝅗𝅥}\rangle]) \end{aligned}$$

Where $\textbf{next}^n \equiv \overbrace{\textbf{next}\,(\textbf{next}\,(\ldots}^{n\,\text{times}}$ Each bar corresponds to a time unit and posts its relevant information to the store. The last note(action) is replicated in order to eliminate states with no notes been played, the interpretation of such state in logic can be complex so it is avoided. A user may ask, with the following **CLTL** formula, if at any time the next interval is an octave higher:

$$\Diamond\,(i\,[n_l\,(p)] = b \dot{\rightarrow}\ \circ\,(i\,[n_l\,(p)] = b'))$$

The output of questioning this property is true, as the third bar of the aforementioned period makes an ascending octave. The output of the program is exposed below:

```
The Property to check is:
<>((p1=:59) -> (o(p1=:71)))
---------------------------
The Process to evaluate the property onto is:
(tell(p1=:59) || tell(d1=:75) || tell(p2=:76) || tell(d2=:75) ||
next((tell(p1=:59) || tell(d1=:75) || tell(p2=:76) || tell(d2=:50)))
|| next^2((tell(p1=:71) || tell(d1=:75) || tell(p2=:86) ||
tell(d2=:75))) || !(next^3((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200)))))
---------------------------
The Kripke Structure of the process is:
---Node 1:---
```

```
Initial:true Normal:[] Temporal:[next^2((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) ||
tell(d1=:75) || tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) ||
tell(d1=:75) || tell(p2=:86) || tell(d2=:75))) (tell(p1=:59) ||
tell(d1=:75) || tell(p2=:76) || tell(d2=:50)) ] Store:[d2=:75 p2=:76
d1=:75 p1=:59 ]
---Node 2:---
Initial:false Normal:[] Temporal:[(tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200)) next^2((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) ] Store:[d2=:75 p2=:86 d1=:75
p1=:71 ]
---Node 3:---
Initial:false Normal:[] Temporal:[(tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200)) next^2((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) ] Store:[d2=:200 p2=:86 d1=:75
p1=:71 ]
---Node 4:---
Initial:false Normal:[] Temporal:[(tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:75)) next^2((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) ] Store:[d2=:50 p2=:76 d1=:75
p1=:59 ]
---Edges---
1#[4]
2#[3]
3#[3]
4#[2]
----------------------------
The property is... true
```

Second property

Rhythmic properties can also be worked on the proposed method along the construction of non-linear and execution-dependant processes. The second property works also as an example on how to achieve this.

Prokofiev uses a specific rhythm cell to achieve the sense of chord cadences, he also provides a rhythmic resolution to this cell in the form of liquidation. Figure 5.2 shows this particular pattern and two forms in which Prokofiev liquidates it. Note that in both liquidation bars the final duration are quarter(♩) and half(𝅗𝅥) notes respectively.

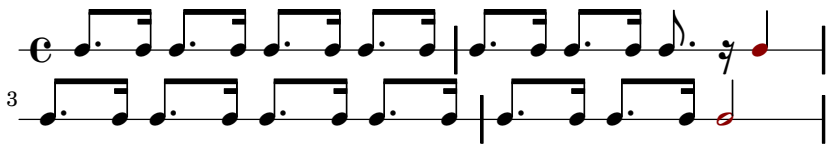


Figure 5.2: Rhythmic cell and its resolutions in red color

It is of significant importance the way the melody develops after these rhythmic resolutions. Let *dur* be the duration of the resolution, the ANTESCOFO score in appendix D.2 captures this musical behaviour by constructing the following process:

$$
\begin{aligned}
SCORE_2 \stackrel{\text{def}}{=} & (\textbf{tell}\ dur = \text{♩} + \textbf{tell}\ dur = \text{𝅗𝅥}) \parallel \\
& \textbf{when}\ dur = \text{♩}\ \textbf{do}\,(\textbf{next}\,(!\,\textbf{tell}\ pitch = b) + \textbf{next}\,(!\,\textbf{tell}\ pitch = d)) \parallel \\
& \textbf{when}\ dur = \text{𝅗𝅥}\ \textbf{do}\,(\textbf{next}\,(!\,\textbf{tell}\ pitch = d') + \textbf{next}\,(!\,\textbf{tell}\ pitch = b,))
\end{aligned}
$$

The first and second **when** processes capture how the melody develops if the rhythmic resolution was done by a full beat(♩) and a half note(𝅗𝅥) respectively. For the former case, the melodic information is in bars 5 and 8 and for the latter this information is in bars 7 and 10.

In spite of the fact that the ANTESCOFO score is being followed linearly, the construction of the process has a non-linear correspondence with this order. The construction of the two **when** agents overlap but the correct outcome is reached by using relative delays at the correspondent events that create the choice operators, ultimately triggering the correct order.

For example, in the score the event associated to create a choice point in the note d' after a resolution of a quarter note is delayed at least 2 bars(8 beats).

More so, the final `NTCC` process may be different depending on the actual following of the score. The posting of whether the duration is ♩ or 𝅗𝅥 (**tell** $dur =$ ♩ + **tell** $dur =$ 𝅗𝅥) depends if the actual event was followed. This is achieved through the use of the **@local** error handling scheme in ANTESCOFO.

One may ask if the score always resolves to b or d independently of the way the rhythmic cell was liquidated by the **CLTL** formula:

$$\Box\,(\circ\,(pitch = b \,\dot{\vee}\, pitch = d))$$

This is false and the output of executing the model checking of this property shows some states where this may no be true. The counterexample indicates that the program may reach the pitch b,(an octave lower).

```
The Property to check is:
[](o((p1=:71) v (p1=:74)))
---------------------------
The Process to evaluate the property onto is:
(when(d1=:100 do (!(next(tell(p1=:71))) + !(next(tell(p1=:74))))) ||
when(d1=:200 do (!(next(tell(p1=:86))) + !(next(tell(p1=:59))))) ||
(tell(d1=:100) + tell(d1=:200)))
```

```
--------------------------
The Kripke Structure of the process is:
---Node 1:---
Initial:true Normal:[] Temporal:[tell(p1=:59) !(next(tell(p1=:59))) ]
Store:[d1=:200 ~(d1=:100) ]
---Node 2:---
Initial:false Normal:[] Temporal:[tell(p1=:86) !(next(tell(p1=:86))) ]
Store:[p1=:86 ]
---Node 3:---
Initial:true Normal:[] Temporal:[tell(p1=:74) !(next(tell(p1=:74))) ]
Store:[d1=:100 ]
---Node 4:---
Initial:false Normal:[] Temporal:[tell(p1=:71) !(next(tell(p1=:71))) ]
Store:[p1=:71 ]
---Node 5:---
Initial:true Normal:[] Temporal:[tell(p1=:71) !(next(tell(p1=:71))) ]
Store:[d1=:100 ]
---Node 6:---
Initial:false Normal:[] Temporal:[tell(p1=:74) !(next(tell(p1=:74))) ]
Store:[p1=:74 ]
---Node 7:---
Initial:true Normal:[] Temporal:[tell(p1=:86) !(next(tell(p1=:86))) ]
Store:[d1=:200 ~(d1=:100) ]
---Node 8:---
Initial:false Normal:[] Temporal:[tell(p1=:59) !(next(tell(p1=:59))) ]
Store:[p1=:59 ]
---Edges---
1#[8]
2#[2]
3#[6]
4#[4]
5#[4]
6#[6]
7#[2]
8#[8]
--------------------------
The property is... false
A counter example is in the following trace of the store:
---State 8, ID 32---
Store:
p1=:59
---State 8, ID 38---
Store:
p1=:59
---State 8, ID 34---
Store:
p1=:59
---State 8, ID 36---
Store:
p1=:59
---State 8, ID 40---
Store:
p1=:59
---State 8, ID 30---
Store:
p1=:59
---Edges---
32#[32 34 36 30]
38#[38 40]
34#[38 40]
36#[32 34 36 30]
40#[32 34 36 30]
30#[38 40]
```

Figure 5.3: On-beat notes marked in color red

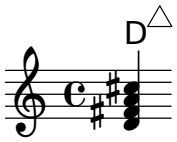
Third property

The final property concerns the harmonic properties of the *on-beats* on the first half of the excerpt. The *on-beats* are the rhythmic accents inherent to each bar, they have a special role in guiding the harmony of tonal music because their occurrence divides the bar time in the same exact duration giving the sense of progression. Figure 5.3 exemplifies this concept.

The score in appendix D.3 captures the value of the on-beats at the next bar whenever the last note takes a full beat establishing a melodic center. This happens in bars 4 and 7, and the `NTCC` process created throughout the score reflects whenever this full beat is the dominant or not(b is the dominant of e). Let on_n be the n-th on-beat of a bar, the process is:

$$SCORE_3 \stackrel{\text{def}}{=} !\,\textbf{when}\ on_1 = b\ \textbf{do next}\ (\textbf{tell}\ (on_1 = b \wedge on_2 = f\sharp \wedge on_3 = d \wedge on_4 = f\sharp)) \parallel \\ !\,\textbf{unless}\ on_1 = b\ \textbf{do next}\ (\textbf{tell}\ (on_1 = d \wedge on_2 = d \wedge on_3 = c\sharp \wedge on_4 = d))$$

The property of temporarily modulating to its parallel(G major) through the dominant chord:



could be encoded by asking the whether the on-beats entail this chord:

$$\Diamond\,(on_1 = d \mathbin{\dot{\wedge}} on_3 = c\sharp)$$

The note $c\sharp$ is very important to the assumption that Prokofiev is modulating because this note leaves the preset melodic line, however the other notes from the dominant are already present in the melody. This property is true and is even more evident in bars 9 and 10 where there is some tonal ambiguity before reiterating the main theme in bar 11, thus going back to the initial tonality. The output of the program is below:

```
The Property to check is:
<>((p1=:59) -> (o(p1=:71)))
---------------------------
The Process to evaluate the property onto is:
(tell(p1=:59) || tell(d1=:75) || tell(p2=:76) || tell(d2=:75) ||
next((tell(p1=:59) || tell(d1=:75) || tell(p2=:76) || tell(d2=:50)))
|| next^2((tell(p1=:71) || tell(d1=:75) || tell(p2=:86) ||
tell(d2=:75))) || !(next^3((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200)))))
---------------------------
The Kripke Structure of the process is:
---Node 1:---
Initial:true Normal:[] Temporal:[next^2((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) ||
tell(d1=:75) || tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) ||
tell(d1=:75) || tell(p2=:86) || tell(d2=:75))) (tell(p1=:59) ||
tell(d1=:75) || tell(p2=:76) || tell(d2=:50)) ] Store:[d2=:75 p2=:76
d1=:75 p1=:59 ]
---Node 2:---
Initial:false Normal:[] Temporal:[(tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200)) next^2((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) ] Store:[d2=:75 p2=:86 d1=:75
p1=:71 ]
---Node 3:---
Initial:false Normal:[] Temporal:[(tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200)) next^2((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) ] Store:[d2=:200 p2=:86 d1=:75
p1=:71 ]
---Node 4:---
Initial:false Normal:[] Temporal:[(tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:75)) next^2((tell(p1=:71) || tell(d1=:75) ||
tell(p2=:86) || tell(d2=:200))) !(next^3((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200)))) next((tell(p1=:71) || tell(d1=:75)
|| tell(p2=:86) || tell(d2=:200))) ] Store:[d2=:50 p2=:76 d1=:75
p1=:59 ]
---Edges---
1#[4]
2#[3]
3#[3]
4#[2]
---------------------------
The property is... true
```

5.2 Discussion and Contributions

The work presented herein has proposed a scheme of multimedia interaction for the NTCC language in the form of representing and proving musical properties. The main method consists of using the ANTESCOFO framework and a protocol specialized in multimedia for translating a real-life audio signal to a NTCC process. Afterwards, a musical property in the form of a **CLTL** formula is fed into the model checker module and the output corresponds to the truth value of the

proposed requirement.

The main contributions of the present work are summarized as follows: (1) a general scheme for proving musical properties in tonal music using ANTESCOFO, (2) a multi-platform thread-safe Mozart/Oz implementation of the OSC(Both Client and Server sides) multimedia protocol as well as a Client/Server implementation of the *Minuit* protocol and finally an updated *Minuit* client implementation in Pure Data, (3) a model checking library for the `NTCC` implemented in Oz with support for `NTCC` process and **CLTL** formula construction

The last two contributions are specific entries in the tasks of the `REACT+` project. This last section then completes the document and offers some concluding remarks and ideas of future lines of work.

5.2.1 Conclusions

From the solution design and implementation in chapter 4 and the experiments and results in this chapter the author can conclude:

- ANTESCOFO supplies an efficient and easy to use platform to score align an input signal and trigger events according to a set of rules dependent on time and data presence. This is specially applicable if the musical score is known a priory and the alignment process serves as director of another related process.
- Using a multimedia protocol for communication of two unrelated frameworks is a convenient choice if emphasis is put on high level abstraction of the data and ease of use. This is true provided that the nature of data is from multimedia applications and security nor throughput efficiency are priorities.
- Careful attention is to be paid to translating the relevant aspects of a score execution to the `NTCC` process in order to truly capture the real behaviour of the musical piece under study and correctly form the process. A choice would be to translate *every* aspect of the input but this is sometimes impractical and not necessary, specially in long pieces.
- **LTL** formulas provide a straightforward form to express properties about timed events. About the musical aspects(melody, harmony), the **CLTL** extension offers constraints as the expressive tool to reason about these phenomena.
- The `NTCC` calculus offers an expressive way to construct models related to music behaviour, this is more evident as timed events and non-deterministic executions are part of the target models. However, the not deterministic duration of a time unit may cause problems if reasoning on the direct relation

of time in a musical piece and time in an NTCC process. This is a key aspect to be taken into account if the music to NTCC translation is to be somehow formalized.

5.2.2 Future Work

Finally, here are some proposed lines of future work directly and indirectly attached to this project:

(a) As mentioned before, the musical event to NTCC agent translation in all experiments was conceived in an empiric fashion without a fixed set of rules. A future project would be to formalize this translation in a consistent but not necessarily deterministic form, keeping in mind that the conciliation of NTCC time and musical time is a prime milestone.

(a) The method proposed here checks fixed properties and only after the whole NTCC process is ultimately constructed. A feedback method could be devised in order to affect the musical input according to the output of the verification phase.

(a) Linked to the last idea, this feedback might be at a constant rate and/or recursively shaping the musical input for the scheme to work something *on the fly*. For this to be implemented, a good starting point is to dwell into incremental model checking for NTCC .

(a) Lastly, the whole method could benefit using the latest version of ANTESCOFO where user-defined synchronization and error handling strategies are possible for the scheme to have more flexible ways of translating music to NTCC agents.

Bibliography

[AAl+10] A.Allombert et al. "VIRAGE : Designing an interactive intermedia sequencer from users requirements and theoretical background". In: *ICMC 2010 (International Computer Music Conference), June 1-5, New York, USA* (2010).

[AG05] Luca Aceto and Larsen Kim G. *An introduction to Milner's CCS*. 2005. URL: http://www.cs.auc.dk/luca/SV/intro2ccs.pdf.

[AG99] M. Abadi and A. D. Gordon. "A calculus for cryptographic protocols: The Spi calculus". In: *Information and Computation* (1999), pp. 1–70.

[AGP06] A. Arbeláez, J. Gutiérrez, and J.A. Pérez. "Timed concurrent constraint programming in systems biology". In: *Newsletter of the ALP* 19.4 (2006).

[AGV06] M. Alpuente, B. Gramlich, and A. Villanueva. "A Framework for Timed Concurrent Constraint Programming with External Functions". In: *Electronic Notes in Theoretical Computer Science* (2006).

[Aks+12] S. Akshay et al. *Overview of Robustness in Timed Systems*. http://anr-impro.irccyn.ec-nantes.fr/deliverables/ImpRo-D21.pdf. 2012.

[Alp+06] María Alpuente et al. "Verifying Real-Time Properties of tccp Programs." In: *J. UCS* 12.11 (2006), pp. 1551–1573.

[And12] Torsten Anders. *OSC Implementation in Strasheela*. http://strasheela.sourceforge.net/strasheela/contributions/anders/OSC/doc/. 2012.

[Ari12] Jaime Arias. *Model Checking for the TCC Calculus*. Bachelor Thesis. Dec. 2012.

[Ass82] MIDI Manufacturers Association. *MIDI Specification 1.0*. http://www.midi.org/techspecs/midispec.php. 1982.

[AVI06] AVISPA. *REACT(Robust Theories for Emerging Applications in Concurrency Theory*. http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:react. 2006.

[AVI12] AVISPA. *REACT+(Robust Theires for Emerging Applications in Concurrency Theory: Processes and Logic used in Emergent Systems)*. http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:react-plus. 2012.

[BCH12] Sandie Balaguer, Thomas Chatain, and Stefan Haar. *Concurrent Semantics of Timed Distributed Systems*. http://anr-impro.irccyn.ec-nantes.fr/deliverables/ImpRo-D41.pdf. 2012.

[Ben10] Ross Bencina. *OSCPack*. http://www.rossbencina.com/code/oscpack. 2010.

[CC92] P. Cousot and R. Cousot. "Abstract interpretation and application to logic programs". In: *The Journal of Logic Programming* 13.2 (1992), pp. 103–179.

[CG00] Luca Cardelli and Andrew Gordon. "Mobile Ambients". In: *Theoretical Computer Science* (2000), pp. 177–213.

[Cha+05] Philippe Charles et al. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: *OOPLSA '05*. 2005.

[CNM12] CNMAT. *OSC Implementations*. http://opensoundcontrol.org/implementations. 2012.

[Con08a] Arshia Cont. "ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music". In: *Proceedings of International Computer Music Conference (ICMC)*. Belfast, Aug. 2008. URL: http://articles.ircam.fr/textes/Cont08a/.

[Con08b] Arshia Cont. "Modeling Musical Anticipation: From the time of music to the music of time". PhD thesis. University of Paris 6 and University of California in San Diego, Oct. 2008. URL: http://cosmal.ucsd.edu/arshia/thesis/.

[Con10] Arshia Cont. "A coupled duration-focused architecture for realtime music to score alignment". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.6 (2010), 974–987. URL: http://articles.ircam.fr/textes/Cont09a/.

[Con11] Arshia Cont. "On the creative use of score following and its impact on research". In: *Sound and Music Computing*. Padova, Italy, July 2011. URL: http://articles.ircam.fr/textes/Cont11a/.

[Ech11] José Echeveste. *Stratégies de synchronisation et gestion des variables pour l'accompagnement musical automatique*. 2011.

[Ech+12] José Echeveste et al. "Antescofo: a Domain Specific Language for real time musician-computer interaction". To appear. Paris, France, Apr. 2012.

[FS09] Adrian Freed and Andy Schmeder. "Features and Future of Open Sound Control version 1.1 for NIME". In: *NIME*. Apr. 2009. URL: http://cnmat.berkeley.edu/node/7002.

[Gar08] H. Garavel. "Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular". In: *Proceedings of LIX Colloquium on Emergent Trends in Concurrency Theory* (2008), pp. 149–164.

[GJS02] V. Gupta, R. Jagadeesan, and V.A. Saraswat. "Truly concurrent constraint programming". In: *Theoretical computer science* 278.1 (2002), pp. 223–255.

[Gro13] AVISPA Group. *ntccMC: Bounded-time automata-based model checker for NTCC*. http://ntccmc.sourceforge.net/. 2013.

[Gut+07] J. Gutiérrez et al. "Timed concurrent constraint programming for analysing biological systems". In: *Electronic Notes in Theoretical Computer Science* 171.2 (2007), pp. 117–137.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[Her+11] D. Hermith et al. "Modeling Cellular Signaling Systems: An Abstraction-Refinement Approach". In: *5th International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB 2011)*. Springer. 2011, pp. 321–328.

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: An eternal golden braid*. Vintage, 1979.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[IMS11] International Music Score Library Project IMSLP. *Romeo and Juliet (2nd Suite) - Prokofiev, Sergey*. http://imslp.org/wiki/Romeo_and_Juliet_%282nd_suite%29,_Op.64ter_%28Prokofiev,_Sergey%29. Violins I (with excerpts from suites 1 and 3) (CA). Apr. 2011.

[Kni+12] Sophia Knight et al. “Spatial Information Distribution in Constraint-based Process Calculi”. To appear in CONCUR2012. École Polytechnique, Paris - France, Apr. 2012.

[Lab87] Ericsson Computer Science Laboratory. *Erlang*. http://www.erlang.org/. 1987.

[Lyg04] John Lygeros. “Lecture Notes on Hybrid Systems”. Rio, Patras, GR-26500, Greece, 2004.

[Maz03] Guerino Mazzola. *Topos of Music*. Birkhäuser Basel, 2003.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. The Pitt Building, Trumpington Street, Cambridge, United Kingdom: Cambridge University Press, 1999.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. The MIT Press, 1996. ISBN: 0262133210.

[Nin07] Nintendo. *Super Mario Galaxy*. Nov. 2007.

[NV04] Mogens Nielsen and Frank D Valencia. “Notes on timed concurrent constraint programming”. In: *Lectures on Concurrency and Petri Nets*. Springer, 2004, pp. 702–741.

[OPV07] Carlos Olarte, Catuscia Palamidessi, and Frank Valencia. “Universal Timed Concurrent Constraint Programming”. In: *Logic Programming*. Vol. 4670. Lecture Notes in Computer Science. Springer-Verlag, Aug. 2007, pp. 464–465.

[OR09] C. Olarte and C. Rueda. “A declarative language for dynamic multimedia interaction systems”. In: *Mathematics and Computation in Music* (2009), pp. 218–227.

[ORV13] Carlos Olarte, Camilo Rueda, and Frank D Valencia. “Models and emerging trends of concurrent constraint programming”. In: *Constraints* (2013), pp. 1–44.

[OV08] C. Olarte and F.D. Valencia. “Universal concurrent constraint programing: symbolic semantics and applications to security”. In: *Proceedings of the 2008 ACM symposium on Applied computing*. ACM. 2008, pp. 145–150.

[Pea11] Martin Peach. *OpenSoundControl for PD*. http://puredata.info/downloads/osc. 2011.

[Plo81] G.D. Plotkin. "A Structural Approach to Operational Semantics". In: *Journal of Logic and Algebraic Programming* (1981).

[Pnu77] Amir Pnueli. "The temporal logic of programs". In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57.

[PR08a] Jorge Pérez and Camilo Rueda. "Non-determinism and Probabilities in Timed Concurrent Constraint Programming". In: *Logic Programming*. Ed. by Maria Garcia de la Banda and Enrico Pontelli. Vol. 5366. Lecture Notes in Computer Science. 10.1007/978 – 3 – 540 – 89982 – 2_56. Springer Berlin / Heidelberg, 2008, pp. 677–681. ISBN: 978-3-540-89981-5. URL: http://dx.doi.org/10.1007/978-3-540-89982-2_56.

[PR08b] J. Pérez and C. Rueda. "Non-determinism and probabilities in timed concurrent constraint programming". In: *Logic Programming* (2008), pp. 677–681.

[RAD06] Camilo Rueda, Gérard Assayag, and Shlomo Dubnov. "A Concurrent Constraints Factor Oracle Model for Music Improvisation". In: *Proceedings of Latin American Informatics Conference CLEI 2006* (2006).

[RBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Academic Press, 2006. ISBN: 9780444527264.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. Cambridge, MA, USA: The MIT Press, 2004.

[Sar08] Gerardo Sarria. "Formal Models of Timed Musical Processes". PhD thesis. Cali - Colombia: Universidad del Valle, 2008.

[SJG95] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. "Default timed concurrent constraint programming". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), pp. 272–285.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. "The semantic foundations of concurrent constraint programming". In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1991), pp. 333–352.

[Tac08] Guido Tack. "Gecode: An Open Constraint Solving Library". In: *Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP)* (2008).

[Tor13] Mauricio Toro. *Towards a model checker for ntcc based on Jaime Arias' algorithm on Mozart OZ*. Tech. rep. Universidad Javeriana, Cali, Feb. 2013.

[Val02] Frank Valencia. "Temporal Concurrent Constraint Programming". PhD thesis. University of Aarhus, 2002.

[Val05] Frank D Valencia. "Decidability of infinite-state timed CCP processes and first-order LTL". In: *Theoretical Computer Science* 330.3 (2005), pp. 577–607.

[Vir09a] Virage. *Interfaces pour la communication avec le protocole Minuit*. http://virage.blueyeti.fr/?p=1462. 2009.

[Vir09b] Virage. *Minuit : Propositions pour un système de requête via OSC*. http://virage.blueyeti.fr/?p=1444. 2009.

[WFM03] Matthew Wright, Adrian Freed, and Ali Momeni. "OpenSound Control: state of the art 2003". In: *Proceedings of the 2003 conference on New interfaces for musical expression*. NIME '03. Montreal, Quebec, Canada: National University of Singapore, 2003, pp. 153–160. URL: http://dl.acm.org/citation.cfm?id=1085714.1085751.

[Xen01] Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition*. 2nd. Haromonologia No.6. Pendragon, 2001.

Appendices

Appendix A

Framework Tools

A.1 ANTESCOFO Pure Data Syntax

The ANTESCOFO implementation in Pure Data has a concrete syntax slightly varied from the abstract syntax its authors propose, the table A.1 documents this syntax through examples of each command.

A.2 Minuit Example

The figure A.1 shows the *name-space* of a *minuit* server, the following lines detail an example of an interaction between this *server* and a *client*:

Event/Action	**Result**
`BPM 60.0`	Sets the score tempo to 60 beats per minute.
`NOTE 69 1.0 a4`	A 1 beat note event with midi value 69 and label *a4*.
`CHORD ( 69 72 76 ) 1.5s aminor`	A three-note chord of length 1.5 seconds with label *aminor*.
`TRILL ( 5800 5850 ) 0.5 d_trill_half_semitone`	A half-beat trill on *d*, notes can also be expressed in midi-cents to reach half-semitones. Order does not matter.
`MULTI ( 64 65 66 67 ) 4.0 glide_e_g`	A glissandi from *e* to *g*(order does matter) of 4-beats duration.
`VARIANCE 0.5`	Sensitiveness to pitch detection(factor from 0 to 1 of a semi-tone).
`TEMPO OFF ... TEMPO ON`	Temporally turns tempo decoding OFF/ON, for atemporal events such as *fermatas*.
`0.0 outmsg msg @local`	Attaches an action to the preceding event, the action has a delay of 0 beats and sends the message `msg` to the pure data outlet `outmsg`. The `@local` is the error handling strategy.
`GFWD 1.0 scale_d_crotchet {...}`	Attaches a labeled group of actions delayed to the preceding event, delayed by one beat.
`LFWD 0.0 polyphony 1.5 {...}`	Attaches a labeled loop of a group of actions to the preceding note, the initial execution has no delay and will repeat each 1.5 beats.
`KILL 0.5 polyphony`	Kills the ongoing loop with label `polyphony`, the kill is attached to the preceding event and is delay by half-beat.

`@FUN_DEF beat2msec(X) {`
`| 1000*X*60.0/RT_TEMPO`
`}`
	Creates a function called `beat2msec` with one parameter `X` and returns the calculation inside the body.

`@MACRO_DEF tritone(N,NAME) {`
`| GFWD NAME {`
`| note N 1.0`
`| N+2 1.0`
`| note N+4 1.0`
`| }`
`}`
	Creates a macro of name `tritone` and parameters `N` and `NAME`. Any appearance of `tritone` will unfold into the body of the macro.

Table A.1: ANTESCOFO concrete score language

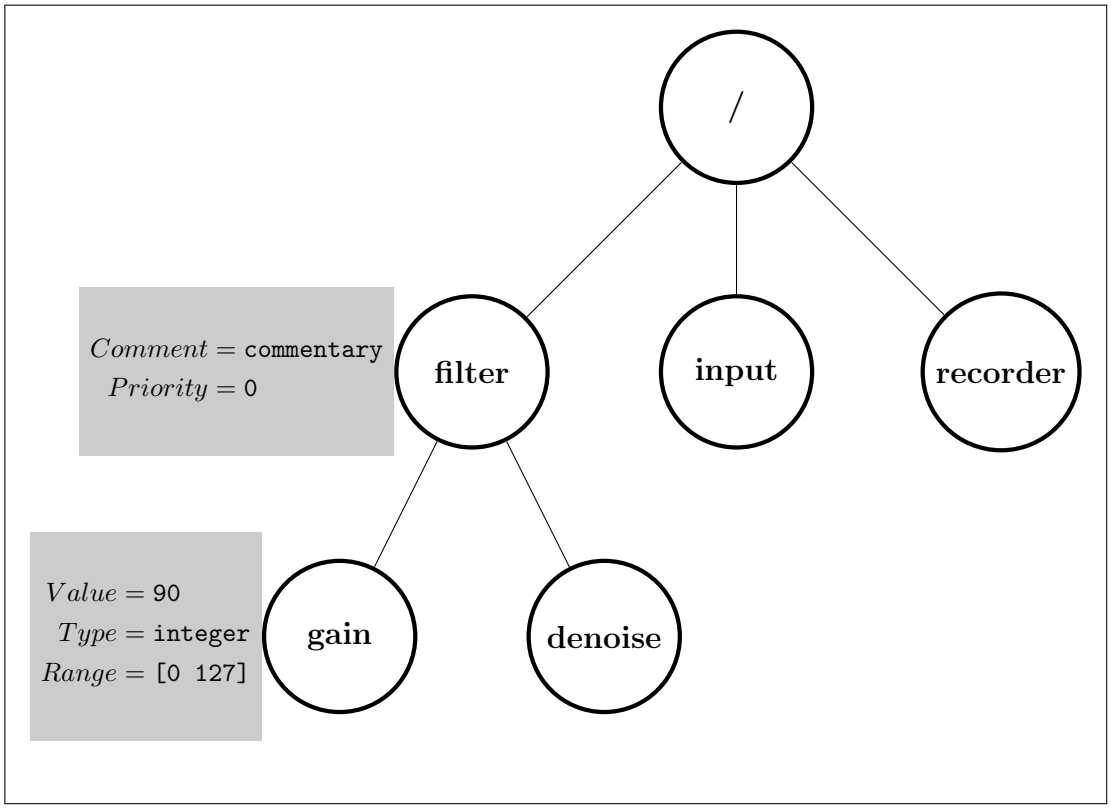


Figure A.1: *Minuit* namespace example

$Client \rightarrow ?namespace/$
$Server \rightarrow : namespace/, sss$ `"filter input recorder" "container container container" ""`

$Client \rightarrow ?namespace/filter$
$Server \rightarrow : namespace/filter, sss$ `"gain denoise" "Data Data" "commentary 0"`

$Client \rightarrow ?namespace/filter/gain$
$Server \rightarrow : namespace/filter/gain, sss$ `"" "" "value type range"`

$Client \rightarrow ?get/filter/gain : value$
$Server \rightarrow : get/filter/gain : value, i$ `90`

$Client \rightarrow ?get/filter/gain : range$
$Server \rightarrow : get/filter/gain : range, ii$ `0 127`

$Client \rightarrow /filter/gain, i$ `91`
$Server \rightarrow NO\ RESPONSE$

$Client \rightarrow ?listen/filter/gain : value$
$Server \rightarrow NO\ RESPONSE$

$Client \rightarrow /filter/gain, i$ `92`
$Client \rightarrow : listen/filter/gain : value, i$ `92`

$Client \rightarrow /filter/gain, i$ `93`
$Client \rightarrow : listen/filter/gain : value, i$ `93`

Appendix B

Multimedia Protocols Documentation

B.1 OSC-Oz

The following is the complete API documentation of the OSC implementation made for Oz. Any transport functionality is excluded and only packing and unpacking of OSC packets is provided for flexibility reasons.

Usage

It is mandatory to include the module at the beginning of any code that uses it so *mozart* can direct the calls to the shared library:

```
functor
import OSC at 'osc-oz.so{native}'
define
    ...
    {OSC.oscFunction ...}
    ...
end
```

Also a copy the platform specific library has to exist in the location specified after the at statement(in the above case the same directory as the oz code).

Packing Functions

'+' stands for input and '?' stands for output.

beginPacket

{OSC.beginPacket *+packetSize ?packetIndex*}

initializes an *OSC packet* with a maximum size of ***packetSize*** bytes. The output ***packetIndex*** returns the index for future manipulation of this packet.

endPacket

{OSC.endPacket *+packetIndex +packetFilename ?packetSize*}

finalizes an ***OSC packet*** identified by ***packetIndex***. The final content of the packet is left in file ***packetFilename*** (atom) of size ***packetSize***, the user is responsible for this pick-up.

beginBundle

{OSC.beginBundle *+packetIndex +secondsTimetag +framesTimetag +typeTimetag*}

initializes a bundle in *OSC packet* identified by ***packetIndex***. The *OSC timetag* of the bundle is constructed from inputs ***secondsTimetag*** and ***framesTimetag*** (each frame is $200 \times 10^{-12} s$). The input ***typeTimetag*** can be the atom 'abs' which makes the timetag an absolute time measure from 0:00am January 1, 1900 or atom 'rel' which makes the timetag a relative time since

the of calling this function. If *secondsTimetag* is 0 and *framesTimetag* is 1 then it means *immediately* regardless of input *typeTimetag*.

endBundle

{OSC.endBundle *+packetIndex*}

finalizes an opened bundle in the *OSC packet* identified by *packetIndex*.

beginMessage

{OSC.beginMessage *+packetIndex +oscAddress*}

begins an *OSC message* in *OSC packet* identified by *packetIndex*. The message has an address pattern specified by input *oscAddress*.

endMessage

{OSC.endMessage *+packetIndex*}

finalizes an opened message in *OSC packet* identified by *packetIndex*.

intArgument

{OSC.intArgument *+packetIndex +intArgument*}

adds an integer argument *intArgument* to a previously opened message in the *OSC packet* identified by *packetIndex*.

floatArgument

{OSC.floatArgument *+packetIndex +floatArgument*}

adds a float argument *floatArgument* to a previously opened message in the *OSC packet* identified by *packetIndex*.

stringArgument

{OSC.stringArgument *+packetIndex +stringArgument*}

adds a string argument *stringArgument* (must be atom) to a previously opened message in the *OSC packet* identified by *packetIndex*.

blobArgument

{OSC.blobArgument *+packetIndex +blobFilename*}

adds a blob argument to a previously opened message in the *OSC packet* identified by *packetIndex*. The blob data is taken from the file *bloblFilename* (must be atom).

Unpacking Functions

dumpPacket

{OSC.dumpPacket *+packetFilename +packetSize ?packetType ?packetIndex*}

dumps an *OSC packet* of size *packetSize* for future processing, the packet content must be in file *packetFilename* (must be atom). The output *packetType* is either 'msg' indicating the packet is a single message or 'bundle' indicating the packet is a bundle composed of more elements.

clearPacket

{OSC.clearPacket *+packetIndex*}

frees allocated memory used by the *OSC packet* identified by ***packetIndex***. Call this when you are no longer inspecting the received packet.

readElement

{OSC.readElement *+packetIndex +bundleIndex +elementIndexIn ?elementType ?elementIndexOut*}

internally reads the ***elementIndexIn***-th element of the *OSC packet* identified by ***packetIndex*** and contained in the bundle ***bundleIndex***. By default if the packet created by dumpPacket is a *bundle*, the first *bundleIndex* is 0. If the output ***elementType*** is the atom 'msg', the element is an *OSC message* which is loaded it into the packetIndex internal structure for future reading. If ***elementType*** is the atom 'bundle' then the element is an *OSC bundle* and in output ***elementIndexOut*** is the *bundleIndex* for future processing.

getElementCount

{OSC.getElementCount *+packetIndex +bundleIndex ?elementCount*}

returns the number of elements of bundle ***bundleIndex*** of *OSC packet* ***packetIndex*** in output ***elementCount***.

getTimeTag

{OSC.getTimeTag *+packetIndex +bundleIndex ?absoluteTime ?relativeTime*}

returns the timetag of bundle ***bundleIndex*** of *OSC packet* ***packetIndex***. Output ***absoluteTime*** (string) specifies this time in seconds since `0:00am January 1, 1900` and output ***relativeTime*** (string) specifies this time in seconds since the call of this function. Either output could also be the string "immediately".

getArgCount

{OSC.getArgCount *+packetIndex ?argumentCount*}

returns the number of arguments the *OSC message* contained in the packet identified by *packetIndex* has. This message must be read before with readElement or dumpPacket.

getMsgArg

{OSC.getMsgArg *+packetIndex +argIndex +blobFilename ?argType ?argValue*}

returns the value of the ***argIndex***-th argument of the *OSC message* contained in the packet identified by ***packetIndex***. Output ***argType*** specifies the type of the argument, this can be:

- 'int': ***argValue*** is the integer argument.
- 'float': ***argValue*** is a string containing the float value of the argument.
- 'string': ***argValue*** is the string argument.
- 'blob': the argument is a blob and it is saved in file ***blobFilename***. ***argValue*** outputs the size of the blob.
- 'other': the argument is of an unrecognized type and ***argValue*** **may** contain the value.

This message must be read before with readElement or dumpPacket.

getMsgAddress

{OSC.getMsgAddress *+packetIndex ?oscAddress*}

returns in ***oscAddress*** the *OSC address* the *OSC message* contained in the packet identified by ***packetIndex*** has, the output is an atom. This message must be read before with readElement or dumpPacket.

B.2 *Minuit* implementation

The following is the specification of all the functions contained in the Oz module implementing the *Minuit* protocol. The transportation layer is excluded from the implementation for flexibility reasons.

Usage

It is mandatory to include the module at the beginning of any code which uses it:

```
functor
import Minuit at 'minuit.ozf'
define
    ...
    Client/Server = {New Minuit.client/server ...}
    {Client/Server minuitFunction(...)}
    ...
end
```

Moreover, a copy of the `'minuit.ozf'` in the source directory as well as the shared library of the *OSC-Oz* implementation have to be present, this is because the *Minuit* protocol is based in the *OSC* protocol.

Server Class

'`+`' stands for input, '`?`' stands for output and `<=` stands for a default value.

```
init
```

{OSC.init *+FileOut<='minuit-out.server' +FileIn<='minuit-in.server' +MsgSize<=1024 +Value<=0 +Range <=[0 127] +Type<="integer" +Priority<=0 +Comment<=''*}

initializes the tree structure(namespace) of the server and creates the base node with the attributes ***Value***, ***Range***, ***Type***, ***Priority*** and ***Comment***. The arguments ***FileOut*** and ***FileIn*** are the temporal placeholders for incoming and outgoing OSC messages with a maximum size of ***MsgSize***.

addNode

{OSC.addNode *+Path +Value<=0 +Range<=[0 127] +Type<="integer"+Priority<=0+Comment<=''*}

adds a node to the server's namespace in the specified ***Path*** with attributes ***Value***, ***Range***, ***Type***, ***Priority*** and ***Comment***. Supported types are "integer", "decimal" and "string". If the ***Path*** is invalid the namespace remains unaltered.

removeNode

{OSC.removeNode *+Path*}

removes the node from the server's namespace in the specified ***Path***, if the ***Path*** is invalid the namespace remains unaltered.

showTree

{OSC.showTree }

pretty prints in the standard output each node from the server's namespace in a depth first order.

processIncomingOSC

```
{OSC.processIncomingOSC }
```

process incoming queued OSC/Minuit messages such as `discover`, `get`/`set` and `listen` queries. It is recommended to call this function within a thread to avoid stalling.

`queueOSCPacket`

```
{OSC.queueOSCPacket +OSCPacket +PacketSize}
```

queue an OSC/Minuit message contained in *`OSCPacket`* of size *`PacketSize`*. If the function `processIncomingOSC` was called and is active then the packet will be processed as soon as it advances through the queue.

`getOSCQueue`

```
{OSC.getOSCQueue $}
```

return the outgoing queue of OSC/Minuit reply messages from the processed queries. This queue is a stream with each element a record of the form `osc(msg: size:)`. The user is responsible for delivering these messages through a transport layer.

Client Class

`init`

```
{OSC.init +FileOut<='minuit-out.client'
+FileIn<='minuit-in.client' +MsgSize<=1024}
```

initializes a client with `FileOut` and `FileIn` as temporal placeholders of the incoming and outgoing OSC messages with a maximum size of *`MsgSize`*.

`discover`

```
{OSC.discover +Path $}
```

queue a *discover* query on the specified ***Path***. This function stalls until a proper reply has been received by the client. The reply is the return value, a record of the form `discover(path: nodes: types: attributes:)` where the *nodes* field is a list containing the children of the argument ***Path***, *types* is a list containing the types of each node respectively and *attributes* is a record of the form `attributes(value: range: type: comment:)` where the node's attributes are contained.

`get`

```
{OSC.get +Path $}
```

queue a *get* query on the specified ***Path***. This function stalls until a proper reply has been received by the client. The reply is the return value, a record of the form `get(path: value:)` where the field *value* contains the value of the node ***Path***.

`set`

```
{OSC.set +Path +Value +Delay<=0.0}
```

queue a *set* operation on the specified ***Path*** with ***Value***. An optional ***Delay*** for this set operation can be entered, if the delay is `0.0` then it means *immediately*(Note: Because Oz only manages signed integers of 32 bits, the maximum fraction of a second currently manageable is $2^{31} \times 200 \times 10^{-12} = 0.4294967$s). A *set* operation does not trigger a response, therefore this function does not stall nor returns anything.

`listen`

```
{OSC.listen +Path +StreamOut}
```

queue a *listen* query on the specified ***Path***. Every time the value of the node in ***Path*** changes, a reply of the form `listen(path: value:)` is appended to the infinite stream ***StreamOut***. The function does not stall nor returns anything.

showReply

{`OSC.showReply` ***+Reply***}

pretty prints in the standard output a received reply from the queries made by the functions `discover`, `get` and `listen`.

processIncomingOSC

{`OSC.processIncomingOSC` }

process incoming queued OSC/Minuit messages such as `discover`, `get`/`set` and `listen` replies. It is recommended to call this function within a thread to avoid stalling.

queueOSCPacket

{`OSC.queueOSCPacket` ***+OSCPacket +PacketSize***}

queue an OSC/Minuit message contained in ***OSCPacket*** of size ***PacketSize***. If the function `processIncomingOSC` was called and is active then the packet will be processed as soon as it advances through the queue.

getOSCQueue

{`OSC.getOSCQueue` ***$***}

return the outgoing queue of OSC/Minuit query messages made by the user with functions `discover`, `get`, `set` and `listen`. This queue is a stream with

each element a record of the form `osc(msg: size:)`. The user is responsible for delivering these messages through a transport layer.

B.2.1 Minuit for Pure Data

An implementation of the client side of the *Minuit* protocol is presented in figure B.1

Figure B.2 presents the basic usage of this *Minuit* client, the input process goes from left to right; first the destination OSC address is provided in the first inlet, then an optional argument or parameter followed by its type are given, after this an optional time-tag can also be provided and finally the user selects the type of query(discover, get, set or listen) before pressing the `bang` widget triggering the send operation.

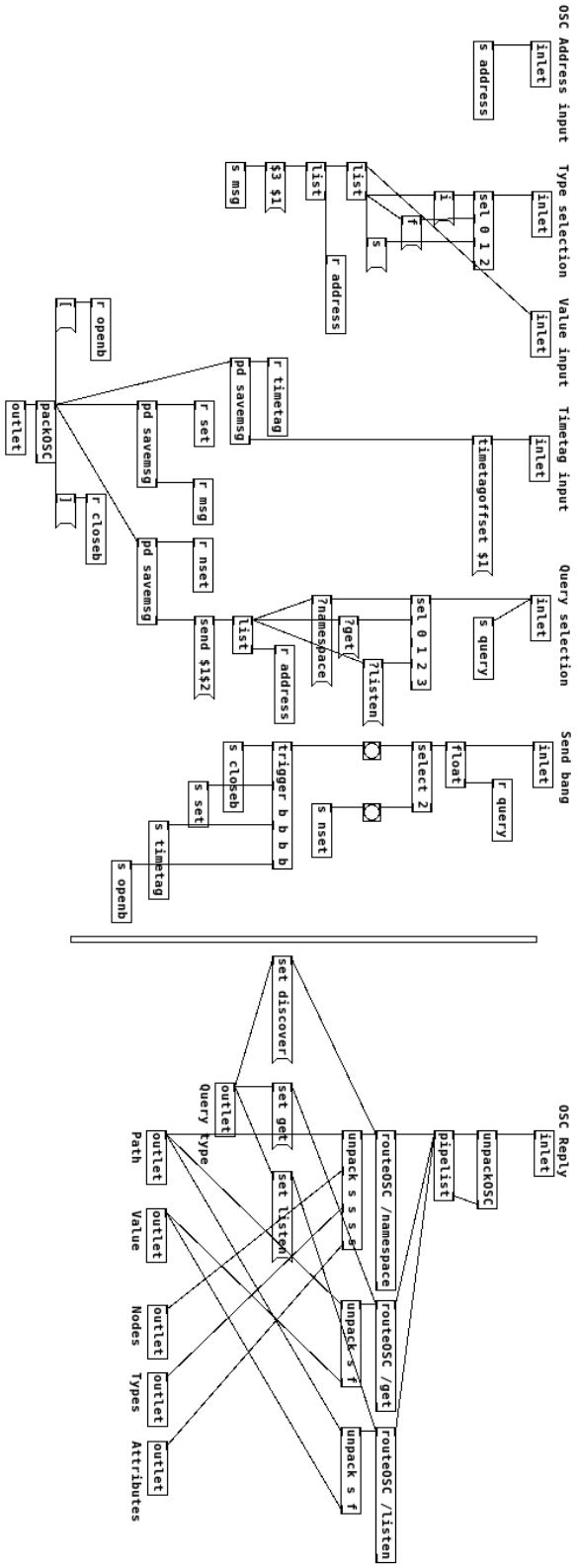


Figure B.1: Minuit Client patch for PD

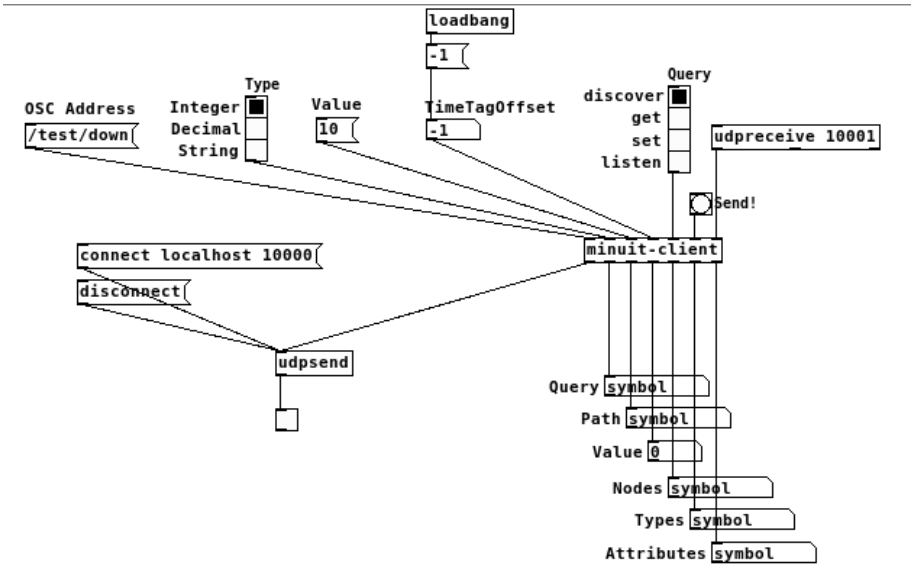


Figure B.2: Minuit Client

Appendix C

Model Checking Documentation

C.1 ntccModelChecker

The following is the specification of the functionality exposed in the model checker module. Presently, there are 4 interfaces implemented: *Linear Time Logic formulae*, *NTCC processes*, *NTCC kripke structures* and *NTCC model checker*.

Usage

It is mandatory to include the module at the beginning of any code that uses it so *mozart* can direct the calls to the pre-compiled library:

```
functor
import NTCC at 'ntccModelChecker.ozf'
define
    Formula = {New NTCC.ltlFormula init( ...  )}
    Process = {New NTCC.ntccProcess init( ...  )  }
    MC = {New NTCC.ntccModelChecker init( Formula Process )
    Answer = {MC modelCheck( $ )}
```

```
end
```

It is a requirement to place the ozf-extension library into the location specified after the at statement(in the above case the same directory as the oz code).

ltlFormula Interface

'<=' stands for default parameter value and '$' means the method produces an output.

```
init
```

{NTCC.ltlFormula init(*LTLFormula <= preposition(constraint: 'x=0'#[x '=:' 0]))*)}

initializes the internal formula with the constructed formula *LTLFormula*. The value of *LTLFormula* is by default the constraint preposition $x = 0$, a constraint is constructed as a pair where the first element is the string representation of the constraint and the second is a triplet of the form [*variable relation value*]. The possible relations are =:, \\=:, >:, <:, >=: and =<:.

```
disjunction
```

{NTCC.ltlFormula disjunction(*$ FormulaLeft<=@formula FormulaRight*)}

outputs a disjunction($\vee$) between *FormulaLeft* and *FormulaRight*, both are constructed LTL formulas. *FormulaLeft* is by default the internal formula.

```
conjunction
```

{NTCC.ltlFormula conjunction(*$ FormulaLeft<=@formula FormulaRight*)}

outputs a conjunction($\wedge$) between *FormulaLeft* and *FormulaRight*, both are constructed LTL formulas. *FormulaLeft* is by default the internal formula.

`implication`

{`NTCC.ltlFormula implication( $ FormulaLeft<=@formula FormulaRight )`}

outputs an implication($\rightarrow$) between *FormulaLeft* and *FormulaRight*, both are constructed LTL formulas. *FormulaLeft* is by default the internal formula.

`negation`

{`NTCC.ltlFormula negation( $ Formula<=@formula )`}

outputs the negation($\neg$) of the constructed LTL formula *Formula*. The internal formula is the default value of *Formula*.

`next`

{`NTCC.ltlFormula next( $ Formula<=@formula )`}

outputs the constructed LTL formula *Formula* being true in the next time unit($\circ$). The default value of *Formula* is the internal formula.

`eventually`

{`NTCC.ltlFormula eventually( $ Formula<=@formula )`}

outputs the constructed LTL formula *Formula* being true eventually in the future($\Diamond$). The default value of *Formula* is the internal formula.

always

{NTCC.ltlFormula always($ *Formula<=@formula*)}

outputs the constructed LTL formula *Formula* being true in all present and future time units($\Box$). The default value of *Formula* is the internal formula.

getFormula

{NTCC.ltlFormula getFormula($)}

outputs the current internal formula.

setFormula

{NTCC.ltlFormula setFormula(*Formula*)}

sets the internal formula to *Formula*.

showFormula

{NTCC.ltlFormula showFormula($ *Formula<=@formula*)}

outputs the constructed LTL formula *Formula* in a human readable form. The default value of *Formula* is the internal formula.

removeSquare

{NTCC.ltlFormula removeSquare($ *Formula<=@formula*)}

outputs the constructed LTL formula *Formula* with every *always* operators($\Box$) removed. Remember that $\Box f \equiv \neg(\Diamond \neg f)$.

`removeDisjunction`

{NTCC.ltlFormula removeDisjunction(*$ Formula<=@formula*)}

outputs the constructed LTL formula *Formula* with every *disjunction* operators($\vee$) removed. Remember that $f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$.

`removeImplication`

{NTCC.ltlFormula removeImplication(*$ Formula<=@formula*)}

outputs the constructed LTL formula *Formula* with every *implication* operators($\rightarrow$) removed. Remember that $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$.

`equalFormulas`

{NTCC.ltlFormula equalFormulas(*$ Formula1 Formula2<=@formula Process<=nil*)}

outputs whether *Formula1* is equal to *Formula2* by using the constraint entailment of the NTCC process *Process*. By default *Formula2* is the internal formula and if *Process* is `nil` then prepositions are compared by list equality.

`calculateClosure`

{NTCC.ltlFormula calculateClosure(*$ Formula<=@formula*)}

outputs the closure of the constructed LTL formula *Formula*. The closure are all the sub-formulas whose truth value can influence the truth value of the formula. By default *Formula* is the internal formula.

getClosure

{NTCC.ltlFormula getClosure($)}

outputs the internal closure.

setClosure

{NTCC.ltlFormula setClosure(*Closure*)}

sets the internal closure to *Closure*.

calculateBasicFormulas

{NTCC.ltlFormula calculateBasicFormulas($ *Closure<=@closure*)}

outputs the basic formulas of the closure *Closure*. Basic formulas are formulas that are satisfied in a specific time unit, mainly prepositions and formulas inside *next* operators. The default value of *Closure* is the internal closure.

getBasicFormulas

{NTCC.ltlFormula getBasicFormulas($)}

outputs the internal basic formulas.

setBasicFormulas

{NTCC.ltlFormula setBasicFormulas(*BasicFormulas*)}

sets the internal basic formulas to *BasicFormulas*.

calculateCombinations

{NTCC.ltlFormula calculateCombinations(*$ Formulas*)}

outputs all possible combinations of the basic formulas *Formulas*.

getCombinations

{NTCC.ltlFormula getCombinations(*$*)}

outputs the internal combinations of some basic formulas.

setCombinations

{NTCC.ltlFormula setCombinations(*Combinations*)}

sets the internal combinations to *Combinations*.

prepositionInCombination

{NTCC.ltlFormula prepositionInCombination(*$ Preposition Combination Process*)}

outputs whether the preposition *Preposition* is entailed by the formulas in the list *Combination* using the constraint entailment of the NTCC process *Process*.

formulaConsistentCombination

{NTCC.ltlFormula formulaConsistentCombination(*$ Formula Combination Process*)}

outputs whether the formula *Formula* is consistent(including entailment of its prepositions) with the list of formulas *Combination* using the constraint entailment of the NTCC process *Process*.

formulaInCombination

{NTCC.ltlFormula formulaInCombination(*$ Formula Combination Process*)}

outputs whether the formula *Formula* is inside the list of formulas *Combination* using the constraint entailment of the NTCC process *Process*.

nonBasicFormulasInCombination

{NTCC.ltlFormula nonBasicFormulasInCombination(*$ Closure<=@closure Combinations<=@combinations Process*)}

outputs the list of formulas *Combinations* plus the non-basic formulas of the closure *Closure* that are consistent with all the formulas from the combination. It uses the constraint entailment of the NTCC process *Process*. By default, the values of *Closure* and *Combinations* are the internal closure and combinations respectively.

ntccProcess Interface

init

{NTCC.ntccProcess init(*Process<=tell([x ':' 0] Variables<=[x] Range<=0#1*)}

initializes the internal process with the constructed NTCC process *Process*. The default value of *Process* is telling the constraint $x = 0$. The constraint is a triplet with the same characteristics as the constraints in the *ltlFormula* interface.

when

{NTCC.ntccProcess when(*$ Constraint<=[x ':' 0] ProcessRight<=@process*)}

outputs the NTCC process *ProcessRight* conditioned by the constraint *Constraint*. *Constraint* defaults to the constraint $x = 0$ and *ProcessRight* defaults to the internal process.

`unless`

```
{NTCC.ntccProcess unless( $ Constraint<=[x ':'  0]
ProcessRight<=@process )}
```

outputs the NTCC process *ProcessRight* conditioned by the absence of the constraint *Constraint*. *Constraint* defaults to the constraint $x = 0$ and *ProcessRight* defaults to the internal process.

`parallel`

```
{NTCC.ntccProcess parallel( $ Process<=@process Processes
)}
```

outputs the NTCC process *Process* in parallel execution with the list of NTCC processes *Processes*. *Process* defaults to the internal process.

`sumation`

```
{NTCC.ntccProcess sumation( $ Process<=@process Processes
)}
```

outputs the NTCC process of a non-deterministic choice between *Process* and the list of NTCC processes *Processes*. *Process* defaults to the internal process.

`next`

```
{NTCC.ntccProcess next( $ Process<=@process )}
```

outputs the NTCC process *Process* where its execution is delayed by one time unit. *Process* defaults to the internal process.

`replication`

```
{NTCC.ntccProcess replication( $ Process<=@process )}
```

outputs the NTCC process *Process* where its execution is replicated on every present and future time units. *Process* defaults to the internal process.

`eventually`

```
{NTCC.ntccProcess eventually( $ Process<=@process )}
```

outputs the NTCC process *Process* where its execution is delayed by a non-deterministic number of time units. *Process* defaults to the internal process.

`getProcess`

```
{NTCC.ntccProcess getProcess( $ )}
```

outputs the internal process.

`setProcess`

```
{NTCC.ntccProcess setProcess( Process )}
```

sets the internal process to *Process*.

`showProcess`

```
{NTCC.ntccProcess showProcess( $ Process<=@process )}
```

outputs the NTCC process *Process* in a human readable form. *Process* defaults to the internal process.

`processToProcedure`

{`NTCC.ntccProcess processToProcedure( $ Process<=@process )`}

outputs the NTCC process *Process* where all its constraints are transformed to Mozart/Oz procedures. *Process* defaults to the internal process.

`constraintEntailment`

{`NTCC.ntccProcess constraintEntailment( $ C1 C2 NotEntailed<=false )`}

outputs whether the constraint *C1* entails (or not entails, depending on the value of *NotEntailed*) the constraint *C2*. The constraints are in the same form of triplets mentioned above.

ntccStructure Interface

`init`

{`NTCC.ntccStructure init( Process )`}

initializes, creates, reduces and renames the kripke structure based off the NTCC process *Process*.

`getStructure`

{`NTCC.ntccStructure getStructure( $ )`}

outputs the internal structure.

setStructure

{NTCC.ntccStructure setStructure(*Structure*)}

sets the internal structure to *Structure*.

showStructure

{NTCC.ntccStructure showStructure(*Structure<=@structure*)}

prints in the standard output the structure *Structure*. By default the value of *Structure* is the internal structure.

createStructure

{NTCC.ntccStructure createStructure(*$ Process*)}

outputs the kripke structure based off the ntcc process *Process*.

reduceStructure

{NTCC.ntccStructure reduceStructure(*$ ExtendedStructure<=@structure*)}

outputs the minimal kripke structure(external transitions only) from the extended kripke structure *ExtendedStructure*. By default the value of *Extended-Structure* is the internal structure.

renameStructure

{NTCC.ntccStructure renameStructure(*$ ReducedStructure<=@structure*)}

outputs the renamed(ordered nodes and edges) kripke structure from the reduced kripke structure *ReducedStructure*. By default the value of *ReducedStructure* is the internal structure.

ntccModelChecker Interface

`init`

{`NTCC.ntccModelChecker init( Formula Process )`}

initializes the model checker object with the LTL goal formula *Formula* and the NTCC base process *Process*.

`getFormula`

{`NTCC.ntccModelChecker getFormula( $ )`}

outputs the internal LTL goal formula.

`getProcess`

{`NTCC.ntccModelChecker getProcess( $ )`}

outputs the internal base NTCC process.

`setFormula`

{`NTCC.ntccModelChecker setFormula( Formula )`}

sets the internal LTL goal formula to *Formula*.

`setProcess`

{NTCC.ntccModelChecker setProcess(*Process*)}

sets the internal NTCC base process to *Process*.

modelCheck

{NTCC.ntccModelChecker modelCheck(*$ Structure<=@structure Formula<=@formula Process<=@process*)}

outputs the model checking of the LTL goal formula *Formula* on the NTCC base process *Process* w.r.t the reduced and renamed kripke structure *Structure*. The default values of *Formula*, *Process* and *Structure* are the internal formula, process and structure. The result is a pair where the first element is `true` or `false` and the second element is a possible trace of the states(counter-example) if the model check bears false.

Appendix D

ANTESCOFO Scores

D.1 First property score

```
@MACRO_DEF AdvanceTime(Delay, Units)
{
    GFWD Delay advance_time_group
    {
        osc msg /ntcc/next Units
        osc msg /ntcc/par
    }
}

@MACRO_DEF TellNote1(Delay, Pitch, Duration)
{
    GFWD Delay tell_note_group
    {
        osc msg /ntcc/tell p1 =: Pitch
        osc msg /ntcc/tell d1 =: Duration
    }
}

@MACRO_DEF TellNote2(Delay, Pitch, Duration)
{
    GFWD Delay tell_note_group
    {
        osc msg /ntcc/tell p2 =: Pitch
        osc msg /ntcc/tell d2 =: Duration
    }
}

BPM 100

EVENT 0.0 Start
      ;;; Note definition ;;;
      ; p = Pitch(midi)
      ; d = Duration(1.0 is 1 beat/quarter)
      osc msg /ntcc 0 200 p1 d1 p2 d2
      osc msg /ntcc/par

EVENT 0.0 1st-Bar
```

```
NOTE 59 0.75  Si
     @TellNote1(0.0, 59, 75)
NOTE 64 0.25  Mi
NOTE 67 0.75  Sol
NOTE 71 0.25  Si
NOTE 76 0.75  Mi
     @TellNote2(0.0, 76, 75)
NOTE 71 0.25  Si
NOTE 67 0.75  Sol
NOTE 64 0.25  Mi

EVENT 0.0 2nd-Bar
NOTE 59 0.75  Si
     GFWD 0.0 Group
     {
        0.0 @AdvanceTime(0.0,1)
        0.0 @TellNote1(0.0, 59, 75)
     }
NOTE 64 0.25  Mi
NOTE 67 0.75  Sol
NOTE 71 0.25  Si
NOTE 76 0.5   Mi
     GFWD 0.0 Group
     {
        0.0 @TellNote2(0.0, 76, 50)
        0.0 osc msg /ntcc/par
     }
NOTE 0 0.5    Rest
NOTE 71 1.0   Si

EVENT 0.0 3rd-Bar
NOTE 71 0.75 Si
     GFWD 0.0 Group
     {
        0.0 @AdvanceTime(0.0,2)
        0.0 @TellNote1(0.0, 71, 75)
     }
NOTE 74 0.25 Re
NOTE 78 0.75 Fa#
NOTE 83 0.25 Si
NOTE 86 0.75 Re
     GFWD 0.0 Group
     {
        0.0 @TellNote2(0.0, 86, 75)
        0.0 osc msg /ntcc/par
     }
NOTE 83 0.25 Si
NOTE 78 0.75 Fa#
NOTE 74 0.25 Re

EVENT 0.0 4th-Bar
NOTE 71 0.75  Si
     GFWD 0.0 Group
      {
        0.0 osc msg /ntcc/rep
        0.0 @AdvanceTime(0.0,3)
        0.0 @TellNote1(0.0, 71, 75)
      }
NOTE 74 0.25  Re
NOTE 78 0.75  Fa#
NOTE 83 0.25  Si
NOTE 86 2.0   Re
     GFWD 0.0 Group
```

```
     {
        0.0 @TellNote2(0.0, 86, 200)
        0.0 osc msg /ntcc/par
     }

EVENT 0.0 5th-Bar
      osc msg /ntcc/par
NOTE 86 0.75 Re
NOTE 87 0.25 Mib
NOTE 86 0.75 Re
NOTE 85 0.25 Do#
NOTE 74 1.0  Re
NOTE 86 1.0  Re

EVENT 0.0 6th-Bar
NOTE 74 0.75 Re
NOTE 75 0.25 Mib
NOTE 74 0.25 Re
NOTE 73 0.25 Do#
NOTE 73 0.25 Do#
NOTE 73 0.25 Do#
NOTE 73 1.0  Do#
NOTE 62 1.0  Re

EVENT 0.0 7th-Bar
NOTE 62 0.75 Re
NOTE 63 0.25 Mib
NOTE 62 0.75 Re
NOTE 61 0.25 Do#
NOTE 61 0.75 Do#
NOTE 63 0.25 Mib
NOTE 62 0.75 Re
NOTE 61 0.25 Do#

EVENT 0.0 8th-Bar
NOTE 61 0.75 Do#
NOTE 67 0.25 Sol
NOTE 66 2.0  Fa#
NOTE 59 1.0  Si

EVENT 0.0 9th-Bar
NOTE 59 0.75 Si
NOTE 64 0.25 Mi
NOTE 67 0.75 Sol
NOTE 71 0.25 Si
NOTE 76 0.75 Mi
NOTE 71 0.25 Si
NOTE 67 0.75 Sol
NOTE 64 0.25 Mi

EVENT 0.0 10th-Bar
NOTE 59 0.75 Si
NOTE 64 0.25 Mi
NOTE 67 0.75 Sol
NOTE 71 0.25 Si
NOTE 76 0.5  Mi
NOTE 0 0.5   Rest
NOTE 71 1.0  Si

EVENT 0.0 11th-Bar
NOTE 71 0.75 Si
NOTE 74 0.25 Re
NOTE 78 0.75 Fa#
```

```
NOTE 83 0.25 Si
NOTE 86 0.75 Re
NOTE 83 0.25 Si
NOTE 78 0.75 Fa#
NOTE 74 0.25 Re

EVENT 0.0 12th-Bar
NOTE 71 0.75 Si
NOTE 74 0.25 Re
NOTE 78 2.0  Fa#
NOTE 78 1.0  Fa#

EVENT 0.0 13th-Bar
NOTE 78 0.75 Fa#
NOTE 79 0.25 Sol
NOTE 78 0.25 Fa#
NOTE 77 0.25 Mib
NOTE 77 0.25 MIb
NOTE 77 0.25 Mib
NOTE 77 1.0  Mib
NOTE 66 1.0  Fa#

EVENT 0.0 14th-Bar
NOTE 65 0.75 Mib
NOTE 67 0.25 Sol
NOTE 66 0.75 Fa#
NOTE 82 0.25 La#
NOTE 77 1.0  Mi#
NOTE 66 1.0  Fa#
```

D.2 Second property score

```
@MACRO_DEF AdvanceTime(Delay, Units)
{
    GFWD Delay advance_time_group
    {
        0.0 osc msg /ntcc/next Units
    }
}

@MACRO_DEF TellNote(Delay, Pitch)
{
    GFWD Delay tell_note_group
    {
        osc msg /ntcc/tell p1 =: Pitch
    }
}

BPM 100

EVENT 0.0 Start
      ;;; Note definition ;;;
      ; p = Pitch(midi)
      ; d = Duration(1.0 is 1 beat/quarter)
      osc msg /ntcc 0 200 p1 d1
      osc msg /ntcc/par

EVENT 0.0 1st-Bar
NOTE 59 0.75  Si
```

```
NOTE 64 0.25  Mi
NOTE 67 0.75  Sol
NOTE 71 0.25  Si
NOTE 76 0.75  Mi
NOTE 71 0.25  Si
NOTE 67 0.75  Sol
NOTE 64 0.25  Mi

EVENT 0.0 2nd-Bar
NOTE 59 0.75  Si
NOTE 64 0.25  Mi
NOTE 67 0.75  Sol
NOTE 71 0.25  Si
NOTE 76 0.5   Mi
NOTE 0 0.5    Rest
NOTE 71 1.0   Si
     GFWD 0.0 Group
     {
        0.0 osc msg /ntcc/when d1 =: 100
        0.0 osc msg /ntcc/sum
        34.0 osc msg /ntcc/tell d1 =: 100 @local
     }

EVENT 0.0 3rd-Bar
NOTE 71 0.75 Si
     GFWD 0.0 Group
     {
        0.0 osc msg /ntcc/rep
        0.0 @AdvanceTime(0.0,1)
        0.0 @TellNote(0.0,71)
     }
NOTE 74 0.25 Re
NOTE 78 0.75 Fa#
NOTE 83 0.25 Si
NOTE 86 0.75 Re
NOTE 83 0.25 Si
NOTE 78 0.75 Fa#
NOTE 74 0.25 Re


EVENT 0.0 4th-Bar
NOTE 71 0.75  Si
NOTE 74 0.25  Re
NOTE 78 0.75  Fa#
NOTE 83 0.25  Si
NOTE 86 2.0   Re
     GFWD 0.0 Group
     {
        8.0 osc msg /ntcc/when d1 =: 200 ; Delay of 2 bars, second when
        0.0 osc msg /ntcc/sum
        24.0 osc msg /ntcc/tell d1 =: 200 @local ; Delay of 6 bars
     }

EVENT 0.0 5th-Bar
NOTE 86 0.75 Re
     GFWD 0.0 Group ;  Delay 2 bars, second when
     {
        8.0 osc msg /ntcc/rep
        0.0 @AdvanceTime(0.0,1)
        0.0 @TellNote(0.0,86)
     }
NOTE 87 0.25 Mib
NOTE 86 0.75 Re
```

```
NOTE 85 0.25 Do#
NOTE 74 1.0  Re
NOTE 86 1.0  Re

EVENT 0.0 6th-Bar
NOTE 74 0.75 Re
     GFWD 0.0 Group
     {
        0.0 osc msg /ntcc/rep
        0.0 @AdvanceTime(0.0,1)
        0.0 @TellNote(0.0,74)
        0.0 osc msg /ntcc/sum
     }
NOTE 75 0.25 Mib
NOTE 74 0.25 Re
NOTE 73 0.25 Do#
NOTE 73 0.25 Do#
NOTE 73 0.25 Do#
NOTE 73 1.0  Do#
NOTE 62 1.0  Re

EVENT 0.0 7th-Bar
NOTE 62 0.75 Re
NOTE 63 0.25 Mib
NOTE 62 0.75 Re
NOTE 61 0.25 Do#
NOTE 61 0.75 Do#
NOTE 63 0.25 Mib
NOTE 62 0.75 Re
NOTE 61 0.25 Do#

EVENT 0.0 8th-Bar
NOTE 61 0.75 Do#
NOTE 67 0.25 Sol
NOTE 66 2.0  Fa#
NOTE 59 1.0  Si
     GFWD 0.0 Group
     {
        0.0 osc msg /ntcc/rep
        0.0 @AdvanceTime(0.0,1)
        0.0 @TellNote(0.0,59)
        0.0 osc msg /ntcc/sum
     }

EVENT 0.0 9th-Bar
NOTE 59 0.75 Si
         osc msg /ntcc/sum
NOTE 64 0.25 Mi
NOTE 67 0.75 Sol
NOTE 71 0.25 Si
NOTE 76 0.75 Mi
NOTE 71 0.25 Si
NOTE 67 0.75 Sol
NOTE 64 0.25 Mi

EVENT 0.0 10th-Bar
NOTE 59 0.75 Si
NOTE 64 0.25 Mi
NOTE 67 0.75 Sol
NOTE 71 0.25 Si
NOTE 76 0.5  Mi
NOTE 0 0.5   Rest
NOTE 71 1.0  Si
```

```
EVENT 0.0 11th-Bar
NOTE 71 0.75 Si
NOTE 74 0.25 Re
NOTE 78 0.75 Fa#
NOTE 83 0.25 Si
NOTE 86 0.75 Re
NOTE 83 0.25 Si
NOTE 78 0.75 Fa#
NOTE 74 0.25 Re

EVENT 0.0 12th-Bar
NOTE 71 0.75 Si
NOTE 74 0.25 Re
NOTE 78 2.0  Fa#
NOTE 78 1.0  Fa#

EVENT 0.0 13th-Bar
NOTE 78 0.75 Fa#
NOTE 79 0.25 Sol
NOTE 78 0.25 Fa#
NOTE 77 0.25 Mib
NOTE 77 0.25 MIb
NOTE 77 0.25 Mib
NOTE 77 1.0  Mib
NOTE 66 1.0  Fa#

EVENT 0.0 14th-Bar
      osc msg /ntcc/sum
NOTE 65 0.75 Mib
     osc msg /ntcc/par
NOTE 67 0.25 Sol
NOTE 66 0.75 Fa#
NOTE 82 0.25 La#
NOTE 77 1.0  Mi#
NOTE 66 1.0  Fa#
```

D.3 Third property score

```
@MACRO_DEF AdvanceTime(Delay, Units)
{
    GFWD Delay advance_time_group
    {
        0.0 osc msg /ntcc/next Units
        0.0 osc msg /ntcc/par
    }
}

BPM 100

EVENT 0.0 Start
      ;;; Note definition ;;;
      ; p = Pitch(midi)
      ; d = Duration(1.0 is 1 beat/quarter)
      osc msg /ntcc 0 200 on1 on2 on3 on4
      osc msg /ntcc/rep
      osc msg /ntcc/par

EVENT 0.0 1st-Bar
```

```
NOTE 59 0.75  Si
NOTE 64 0.25  Mi
NOTE 67 0.75  Sol
NOTE 71 0.25  Si
NOTE 76 0.75  Mi
NOTE 71 0.25  Si
NOTE 67 0.75  Sol
NOTE 64 0.25  Mi

EVENT 0.0 2nd-Bar
NOTE 59 0.75  Si
NOTE 64 0.25  Mi
NOTE 67 0.75  Sol
NOTE 71 0.25  Si
NOTE 76 0.5   Mi
NOTE 0 0.5    Rest
NOTE 71 1.0   Si
     osc msg /ntcc/when on1 =: 71

EVENT 0.0 3rd-Bar
     @AdvanceTime(0.0,1)
NOTE 71 0.75 Si
     osc msg /ntcc/tell on1 =: 71
NOTE 74 0.25 Re
NOTE 78 0.75 F#
     osc msg /ntcc/tell on2 =: 78
NOTE 83 0.25 Si
NOTE 86 0.75 Re
     osc msg /ntcc/tell on3 =: 86
NOTE 83 0.25 Si
NOTE 78 0.75 F#
     osc msg /ntcc/tell on4 =: 78
NOTE 74 0.25 Re

EVENT 0.0 4th-Bar
      osc msg /ntcc/par
NOTE 71 0.75  Si
NOTE 74 0.25  Re
NOTE 78 0.75  Fa#
NOTE 83 0.25  Si
NOTE 86 2.0   Re

EVENT 0.0 5th-Bar
NOTE 86 0.75 Re
NOTE 87 0.25 Mib
NOTE 86 0.75 Re
NOTE 85 0.25 Do#
NOTE 74 1.0  Re
NOTE 86 1.0  Re
     GFWD 0.0 Group
     {
        0.0 osc msg /ntcc/unless on1 =: 71
        0.0 osc msg /ntcc/next 1
        0.0 osc msg /ntcc/par
     }

EVENT 0.0 6th-Bar
NOTE 74 0.75 Re
     osc msg /ntcc/tell on1 =: 74
NOTE 75 0.25 Mib
NOTE 74 0.25 Re
     osc msg /ntcc/tell on2 =: 74
NOTE 73 0.25 Do#
```

```
NOTE 73 0.25 Do#
NOTE 73 0.25 Do#
NOTE 73 1.0  Do#
     osc msg /ntcc/tell on3 =: 73
NOTE 62 1.0  Re
     osc msg /ntcc/tell on4 =: 62

EVENT 0.0 7th-Bar
      osc msg /ntcc/par
      osc msg /ntcc/par
NOTE 62 0.75 Re
NOTE 63 0.25 Mib
NOTE 62 0.75 Re
NOTE 61 0.25 Do#
NOTE 61 0.75 Do#
NOTE 63 0.25 Mib
NOTE 62 0.75 Re
NOTE 61 0.25 Do#

EVENT 0.0 8th-Bar
NOTE 61 0.75 Do#
NOTE 67 0.25 Sol
NOTE 66 2.0  Fa#
NOTE 59 1.0  Si

EVENT 0.0 9th-Bar
NOTE 59 0.75 Si
NOTE 64 0.25 Mi
NOTE 67 0.75 Sol
NOTE 71 0.25 Si
NOTE 76 0.75 Mi
NOTE 71 0.25 Si
NOTE 67 0.75 Sol
NOTE 64 0.25 Mi

EVENT 0.0 10th-Bar
NOTE 59 0.75 Si
NOTE 64 0.25 Mi
NOTE 67 0.75 Sol
NOTE 71 0.25 Si
NOTE 76 0.5  Mi
NOTE 0 0.5   Rest
NOTE 71 1.0  Si

EVENT 0.0 11th-Bar
NOTE 71 0.75 Si
NOTE 74 0.25 Re
NOTE 78 0.75 Fa#
NOTE 83 0.25 Si
NOTE 86 0.75 Re
NOTE 83 0.25 Si
NOTE 78 0.75 Fa#
NOTE 74 0.25 Re

EVENT 0.0 12th-Bar
NOTE 71 0.75 Si
NOTE 74 0.25 Re
NOTE 78 2.0  Fa#
NOTE 78 1.0  Fa#

EVENT 0.0 13th-Bar
NOTE 78 0.75 Fa#
NOTE 79 0.25 Sol
```

```
NOTE 78 0.25 Fa#
NOTE 77 0.25 Mib
NOTE 77 0.25 MIb
NOTE 77 0.25 Mib
NOTE 77 1.0  Mib
NOTE 66 1.0  Fa#

EVENT 0.0 14th-Bar
NOTE 65 0.75 Mib
NOTE 67 0.25 Sol
NOTE 66 0.75 Fa#
NOTE 82 0.25 La#
NOTE 77 1.0  Mi#
NOTE 66 1.0  Fa#
```