# conga: A Conducting Gesture Analysis Framework

Diplomarbeit
Universität Ulm
Fakultät für Informatik


vorgelegt von

**Ingo Grüll**

1. Gutachter: Prof. Dr. Michael Weber
2. Gutachter: Prof. Dr. Jan Borchers

**April 2005**

## Abstract

Conducting an orchestra is a highly sophisticated art form that matured over centuries. In the last few decades, conducting has also become a form of human-computer interaction, giving conductors different ways to enter conducting input with varying degrees of control and varying levels of success in making the computer perform the way the conductor wants it to.

This diploma thesis describes a framework developed to aid analysis and recognition of conducting gesture input in form of two-dimensional trajectories of the movement of the conductor's baton, or of his right hand, if he is conducting without a baton. The thesis explains the concepts behind the framework, lists some of its characteristic components and gives examples how to use it. In addition, the thesis provides an overview of several computer-based systems that enable their users to conduct musical pieces and briefly looks into other frameworks that have been used to process conducting gesture input in some of the presented systems. It also judges strengths and weaknesses of the framework and mentions how the framework could be extended both for its intended application domain as well as for other application domains.

## Acknowledgements

I would like to express my gratitude to all people who helped me with my work on this paper and the conga framework. In particular, I thank the following persons:

- Rafael “Tico” Ballagas
- Nils Beck
- Jan Borchers
- Jan Buchholz
- Saskia Dedenbach
- Jonathan Diehl
- Linda Goldschmidt
- Britta Grünberg
- Thorsten Karrer
- Henning Kiel
- Jonathan Klein
- Eric Lee
- Teresa Marrin Nakra
- Guido de Melo
- Michael Plichta
- Wolfgang Samminger
- Tanja Scheffold
- Daniel Spelmezan
- Philipp Stephan
- Michael Weber
- Stefan Werner
- Marius Wolf
- Eugen Yu

Last but not least, I also thank my family: my parents Margit and Bolko Grüll and my sister Sascha Grüll.

# Contents

# 1 Introduction

## 1.1 Background

Computers are extremely versatile tools that by now have found a use in nearly every area of our lives. Due to their ever-increasing speed the limiting factor in applying computers to a task often is not the computational performance but rather the interface between computers and their human users. On the input side, mouse and keyboard may be convenient for office work and similar tasks but for a lot of areas better alternatives are needed. Thus researchers examine very different forms of input such as recognition of speech, handwriting or gestures, each with a lot of possible applications and each with its own set of problems. In the field of gesture analysis and recognition, it is often very difficult to discern the meaning of a gesture without analyzing the context in which it occurred. For example, in conversation hand gestures may serve to help understand what is being said. It is more difficult but still possible to understand what is being said without the accompanying hand gestures but it is probably impossible to understand the hand gestures without the accompanying conversation. So gesture recognition in human-computer interaction is most promising if used in combination with other forms of input.

But there are areas where gesture alone is able to convey all the information that is needed. In the area of conducting music, there exists a language of conducting gestures that enable a conductor to communicate with an orchestra. These gestures are more formal than gestures used in everyday human life but they are still very expressive in their own application domain. Consequently, they are an interesting special case of computer gesture input and as a result a significant amount of research has focused on computer-based conducting systems. This thesis also deals with analysis and recognition of conducting gestures. It does not cover other areas of gesture recognition[1].

## 1.2 Motivation

In order to work, computer-based conducting systems need components that process conducting input. When the number of computer-based conducting systems increased and some researchers were involved in the creation of more than one such system, there was not only the need to create these components but also the need to reuse them. As a consequence, some components for processing of conducting input have been implemented using existing frameworks so that they can be used and reused in combination

[1] An in-depth coverage of the complete field of gesture analysis and recognition is well beyond the scope of a single diploma thesis.

with other components contained in or implemented with those frameworks. But there was no framework built specifically for the purpose of conducting gesture analysis and recognition. This diploma thesis describes the framework *conga* which was developed as an input hardware independent toolkit to enable and aid the construction of components that can process, analyze, recognize and track conducting gestures of the conductor's baton or right hand.

## 1.3 Structure of this Thesis

Following this introductory chapter, Chapter 2 presents an overview of existing computer-based conducting systems and introduces the frameworks used in some of those systems. After that, Chapter 3 sketches the context that led to the creation of the conga framework and Chapter 4 describes the design of conga and its components as well as circumstances and problems that influenced this design. Next, Chapter 5 gives examples how the framework can be used and Chapter 6 lists strengths and weaknesses of conga and mentions how it could be extended, in the area of conducting as well as in other application domains. Finally, Appendix A introduces a component that transforms output of conga's gesture tracking components into absolute time values.

# 2 Related Work

Over the last few decades, there have been quite a lot of computer-based systems that take conducting input in one form or the other. This chapter presents an overview of a number of such systems, mostly taken from the area of research. Following the overview it gives an informal comparison of the systems presented in the overview, touching on several important aspects and the different ways the systems handle those aspects. The chapter also includes a section about frameworks that have been used for conducting gesture analysis in some of the systems presented in the overview. Because the frameworks themselves do not place special emphasis on conducting, they are discussed first.

## 2.1 Frameworks Used in Computer-Based Conducting Systems

Three frameworks appear in this section, *LabVIEW*, *EyesWeb* and *Max/MSP*. LabVIEW is the most general and probably the most powerful of the three, while the other two are better suited for developing interactive musical applications and accordingly have often been used in this area. There are, however, some aspects all three frameworks handle similarly.

### 2.1.1 LabVIEW

The company Native Instruments[1] offers *LabVIEW* as "a powerful graphical development environment for signal acquisition, measurement analysis, and data presentation, giving you the flexibility of a programming language without the complexity of traditional development tools" [Ins]. LabVIEW is a rather general framework, suited for, but not aimed specifically at, creating musical applications. It is a very powerful environment with a wide array of predefined modules for signal acquisition and processing and it also features graphic representations for functionality found in normal programming languages like loop constructs or data structures. Programming consists of connecting the icons representing data sources, data containers and functions or modules with wires plus arranging the graphical user interface elements that are part of some modules, showing output and letting the user control the program at runtime. When the program is executed, data flows along the wires. LabVIEW analyzes the data flow and runs structures in parallel, if they are independent of each other. After creating a processing structure in LabVIEW, it can be encapsulated as a module and then used like the built-in modules, hiding the internal structure behind a simple icon. Native Instruments also claims that LabVIEW easily communicates with all kinds of input hardware as well as with other

[1] The website of Native Instruments can be found at http://www.ni.com

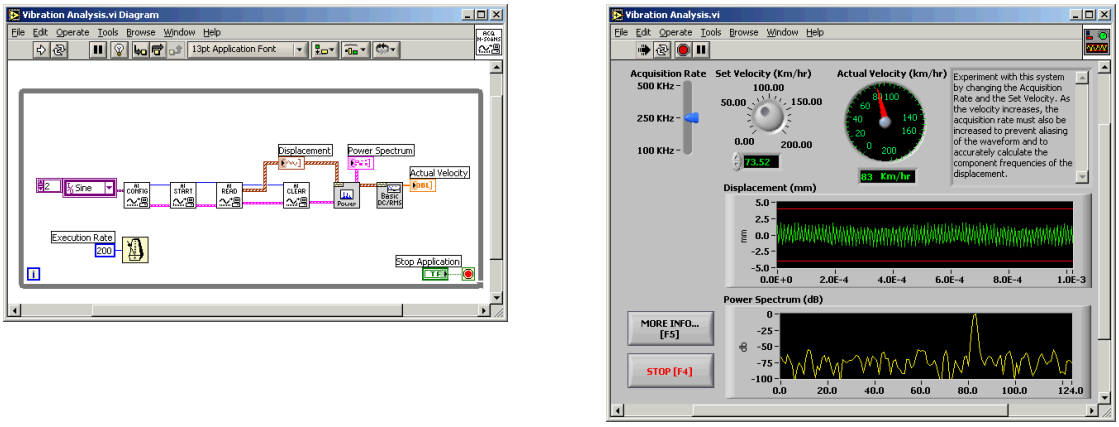


Figure 2.1: Example program in LabView programming environment (image on the left) and its runtime GUI (image on the right). Images taken from "Virtual Instrumentation" tutorial on Native Instruments' website.

software applications. LabVIEW runs on Microsoft Windows, Apple Mac OS X, Linux and Sun Solaris and there are several versions with differing feature sets at different price levels.

### 2.1.2 EyesWeb

One of the projects of InfoMus Lab, the Laboratorio di Informatica Musicale at the University of Genova, Italy, is *EyesWeb*[2] [CCM+04, Vol03]. EyesWeb is an open platform that bears some resemblance to LabView but is more specialized and probably not as powerful: it is a rapid prototyping environment for creating real-time applications in the area of music and multimedia, especially interactive music and dance performances. Video images are EyesWeb's main input, and it provides several computer vision based algorithms for detecting and tracking the bodies, or body parts, of human performers plus their movements, and there are EyesWeb libraries for processing expressive gestures and analyzing two-dimensional trajectories and two-dimensional space [CMV03].

EyesWeb also features a graphical programming environment, where computer vision based modules and simpler processing modules are represented as icons and interconnected with wires. Processing structures created in the graphical programming environment can be encapsulated as modules, but it is also possible to write new modules in C++. For output, EyesWeb can overlay visual representations of extracted features on the input videos, trigger stored video sequences and synthesize new ones, using as parameters information derived from the input data. In addition, EyesWeb can output data via MIDI or over TCP/IP, and one can always write modules in C++ to accommodate special input or output channels. Currently, EyesWeb is available for Microsoft Windows, and a Linux version is planned.

[2]The EyesWeb project website can be found at http://www.eyesweb.org

a) b)

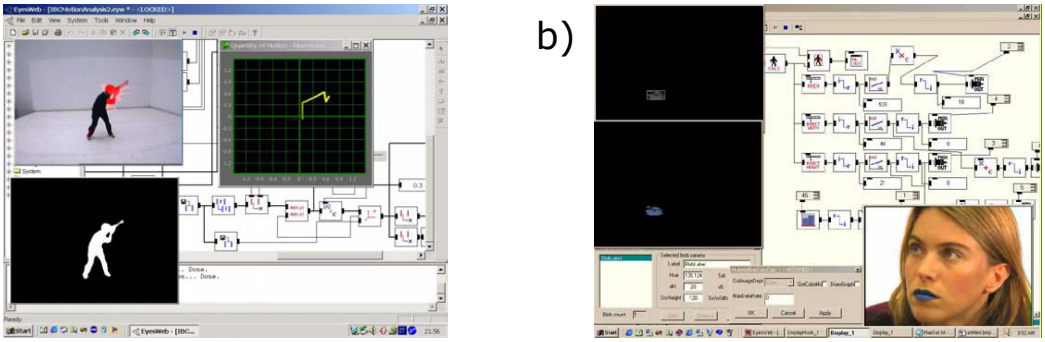

Figure 2.2: a) Gestures being represented as two-dimensional trajectories in EyesWeb. b) EyesWeb patch to process movements of an actress' lips. Images taken from [Vol03].

### 2.1.3 Max/MSP

*Max/MSP* is a product offered for Microsoft Windows and Apple Mac OS X by the company Cycling '74[3]. The "Max" in its name is an homage to Max Mathews. The software was originally written by Miller Puckette at IRCAM, the Institut de Recherche et Coordination Acoustique/Musique in Paris, France.

Figure 2.3: Miller Puckette. Image taken from http://www.crca.ucsd.edu /~msp/x.gif

Max/MSP is a rapid prototyping software for developing real-time music applications, with a graphical programming environment that uses a metaphor derived from patchable modular analog synthesizers and is in some aspects similar to the graphical programming environments of LabView and EyesWeb. Modules like processing units, data containers or system inputs and system outputs are represented as boxes that can be placed on a graphical plane and connected with patch cords. Such an assembly, called a patch, can in turn be encapsulated into a module and then be reused just like any other module. MIDI data, audio signals and control messages are sent along the patch cords, but the order in which data is processed does not only depend on the connections between modules, it depends on the two-dimensional placement of the modules on the plane as well. So if one module output is connected to the inputs of several other modules, the programmer can still determine which of those modules processes the data first. Widgets for user input into a patch or displaying data at runtime are placed on the plane alongside

[3]The website of Cycling '74 can be found at http://www.cycling74.com

processing modules, thereby mixing graphical user interface design with programming. While a patch is being executed, only its widgets are shown. Max/MSP features a lot of basic processing units as well as a lot of signal processing and synthesis oriented ones and Cycling '74 offers an add-on, called *Jitter*, that can handle all kinds of matrix data and is optimized for graphics and video processing, including support for hardware-accelerated OpenGL operations. An SDK is also available, making it possible to write new Max/MSP modules, and there are third parties that offer Max/MSP modules to communicate with special input and output hardware.

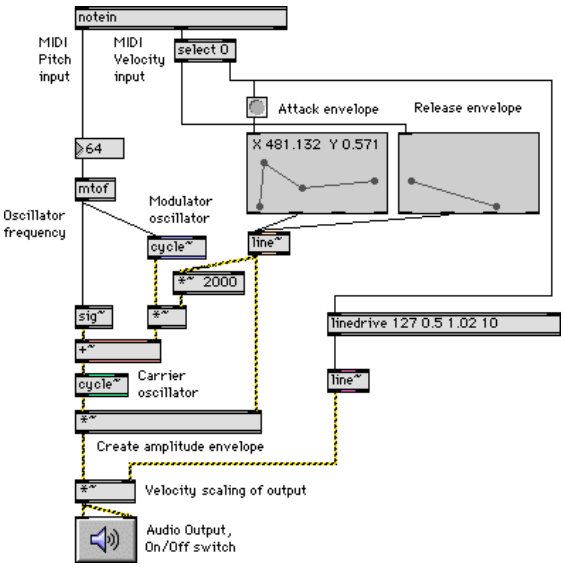


Figure 2.4: A Max/MSP example patch. The image was taken from http://www.cycling74.com /images/msp1.gif

There exist free software packages resembling Max/MSP. One of them, called *Pure Data* and available for Linux, Microsoft Windows and several unix flavors, including Apple Mac OS X, was created and is still maintained by Miller Puckette[4]. Another one, called *jMax* and available for Linux, Microsoft Windows and Apple Mac OS X, is supported by the IRCAM[5].

[4]Pure Data software and documentation can be found at http://www.crca.ucsd.edu/~msp/software.html
[5]jMax software and documentation can be found in the software section of the website of IRCAM at http://www.ircam.fr/institut.html

## 2.2 Overview of Computer-Based Conducting Systems

This overview groups systems by person instead of presenting each system in a separate section. That is, systems appear in the same group if there was the same person in a central role in the development of each of them. Inside each group, systems are ordered chronologically by the year they were created or at least presented to the public. The groups are in chronological order according to the earliest system of each group.

### 2.2.1 Radio Baton and Predecessors

Max Mathews is often called one of the fathers of computer music because of his fundamental work in this field. From 1957 on, he created the *Music* systems (version I to V), software packages for general purpose sound synthesis on a computer [Spo01]. Starting in 1968, Mathews and Moore developed *GROOVE*, a program that enabled the user to compose and edit functions of time interactively and to store them for further use [MM70]. GROOVE initially ran on a computer system equipped with digital-to-analog an analog-to-digital converters. Combined with equipment controlled by voltage like an electronic music synthesizer and input hardware specially built for the system, including a box with buttons, switches and rotary knobs, a keyboard similar to an organ keyboard and a 3D joystick, this system was well suited for sound synthesis.

But Mathews and Moore also had conducting in mind and added the *Conductor Program*, stating that the computer should not act like an instrument used by a player but rather like an orchestra controlled by a conductor: the user of the computer would influence the way a stored score was played by the computer. Input from the input hardware let the Conductor Program advance beat by beat in the score. A drum that had been modified as input device for the system seemed to be more suited to conducting than other input devices, so Mathews developed it into the *Sequential Drum* in 1980.

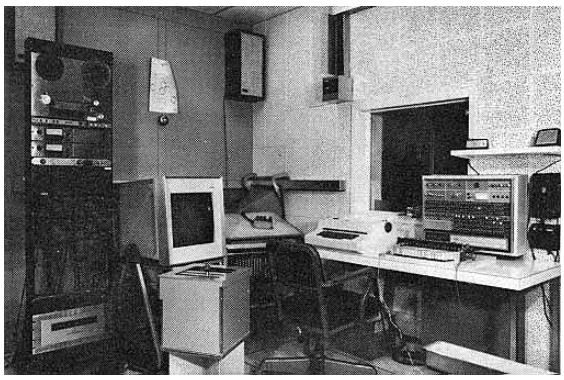

Figure 2.5: Hardware of the GROOVE system. Image taken from [MM70].

The Sequential Drum consisted of a right-angled network of wires and a microphone. The impact of a drum stick brought two wires into contact, giving a two-dimensional position, similar to the way some touch-screens work, and the sound picked up by the microphone indicated how strong the impact was. Unfortunately, strong impacts often caused the wires to break, and between impacts there was no information on the conductor's movements. In order to solve those problems, Mathews and Bob Boies created the *Radio Baton*, a device that is still being used by Mathews and others today [BMS89, BM97]. It consisted of two batons that emitted radio waves from their tips and a rectangular plate equipped with antennas to receive the signals emitted by the batons.

Figure 2.6: Max Mathews and the Radio Baton. Image taken from www.csounds.com /mathews

Those antennas measured the positions of the batons above the plate in three dimensions, but precision of the z coordinate declined with growing distance to the plate. The Radio Baton was connected to a computer via MIDI or a serial interface. A version of the Conductor Program for the Radio Baton operated on prerecorded scores that contained, among other things, triggers linked to notes [Mat00a]. The user conducted with the Radio Baton by working through the triggers defined in the piece. There were two ways for producing trigger events with the Radio Baton. One was to set a value for the z coordinate above the antenna plate and generate a trigger event every time the baton z position crossed the imaginary plane defined by that value. The other was to generate a trigger event if the downward acceleration of a stick decreased to zero after the downward velocity of the stick had increased beyond a velocity threshold [Mat00b]. The Conductor Program provided control over the timing of events in a score with triggers, and control over dynamics, voice balance and timbre with baton positions. To enhance expressive control even further, Johan Sundberg, Anders Friberg, Max Mathews and Gerald Bennett combined the Radio Baton with the *Director Musices* performance grammar, which changes musical expression according to musical context [SFMB01].

### 2.2.2 Conduct System

In 1980, Buxton, Reeves, Fedorkow, Smith and Baecker constructed the *conduct* system [BRF+80]. It was a conducting system based on a microcomputer and included a digital synthesizer as well as a graphics tablet and switches plus sliders as input hardware. The system played prerecorded scores and allowed to change pitch, tempo, articulation, amplitude (volume) and richness (timbre) of notes played by the synthesizer in real time. These parameters were controlled by selecting the desired parameter on screen with the graphics tablet cursor and then changing its value directly, they were not derived from conducting movements of the user.

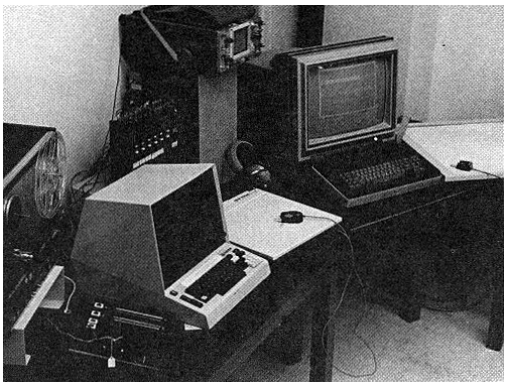

Figure 2.7: User interface hardware of the conduct system. Image taken from [BRF+80].

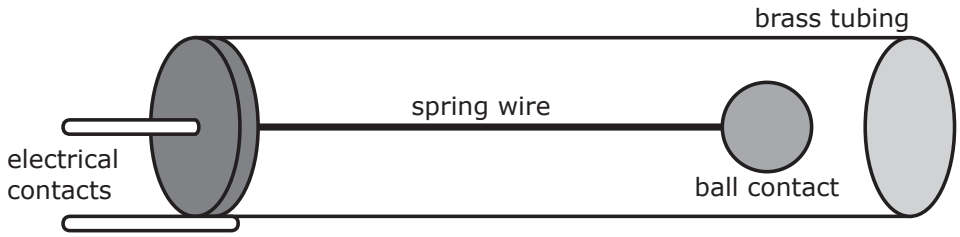


Figure 2.9: Mechanical sensor of the MIDI Baton. Drawing after [KG89].

### 2.2.3 Conductor Follower

Stephen Haflich and Markus Burns presented their *Conductor Follower* in 1983 at the International Computer Music Conference. It extracted beats from the two-dimensional trajectory of a baton, controlling tempo and dynamics of a piece played back by a synthesizer. The baton was a passive device and included no electronics at all. Instead, a corner reflector had been mounted to its tip and ultrasonic range-finders, developed by Polaroid for one of their camera models, were used to track the movement of the baton. This allowed the human conductor to use real conducting gestures to drive the system [Kol04, Sam02].

Figure 2.8: Stephen Haflich using the Conductor Follower. Image taken from [Nak00].

### 2.2.4 MIDI Baton and Successors

In 1989, David Keane and Peter Gross built the *MIDI Baton* because they wanted a computer system that was able to play alongside human performers conducted by a human conductor [KG89]. Their baton could be wielded like a normal baton, but it contained a mechanical sensor that reacted to acceleration. The sensor consisted of a metal ball inside a metal tube, each wired to an electrical contact, as illustrated in Figure 2.9. A spring kept ball and tube separated. If the baton was accelerated strong enough, ball and tube came into contact, generating an electrical signal. The signals from the baton were processed to filter out pulses that should not be regarded as beats and then sent to a sequencer to produce sound. A foot-switch was included in the system to let the conductor start, pause and restart the sequencer. Apart from pulse signals, the MIDI Baton did not provide information about conducting movements, and it only generated timing information and no volume information for the sequencer. There were two successors, the *MIDI Baton II* and the *MIDI Baton III*, but they followed the same basic concept [KW91].

### 2.2.5 Computer Music System that Follows a Human Conductor

Hideyuki Morita, Shuji Hashimoto and Sadamu Ohteru developed their *Computer Music System that Follows a Human Conductor* in 1989 and, together with Hiroshi Watanabe and Tsutomu Harada, they created an enhanced version of the system one year later [MOH89, MWH+90, MHO91]. The first version tracked either a white marker attached to the conductor's baton or the conductor's hand wearing a white glove, using a CCD camera and special feature extraction hardware that passed two-dimensional position values to a personal computer. The computer derived tempo and volume information from upper and lower turning points of the trajectory. The final version used an infrared light source, mounted to the tip of the baton, and a CCD camera with an infrared filter, taking position, velocity and acceleration of the tip of the baton as conducting input. It also added a VPL Research data glove as input device, to be worn on the conductor's left hand. Movements and hand gestures of the left hand were tracked and interpreted by the system to give the conductor more control over the orchestra, for example by selecting an instrument section and then indicating a certain musical expression. The system also included a knowledge database for mapping conducting and hand gestures to musical expression information and it featured a self-evaluation function that enabled the conductor to change the mappings in the knowledge database by telling the system, after he conducted a piece, how good the system had interpreted his conducting. All versions of the system created sound via a MIDI sequencer and MIDI synthesizers.

### 2.2.6 Light Baton

In 1992, Graziano Bertini and Paolo Carosi created the *Light Baton* [BC92]. It was aimed at letting a human conductor conduct musicians in parallel to a computer that played back a prerecorded score. The baton included a battery and a strong LED, which was mounted to its tip. A CCD camera recorded the conductor, who could conduct using normal conducting gestures. A special image acquisition board extracted the light point without using the CPU of the host computer. The prerecorded score was then adjusted according to tempo and volume information derived from the trajectory of the light.

### 2.2.7 Adaptive Conductor Follower and Related Systems

Michael Lee, Guy Garnett and David Wessel built an *Adaptive Conductor Follower* in the year 1992 [LGW92], and Bennett Brecht and Guy Garnett produced an updated version just a year later, which they called *Conductor Follower* [BG95]. A Mattel Power Glove and a Buchla Lightning baton system served the system as input devices. The Buchla Lightning used infrared light to determine a two-dimensional position of its baton and passed this position via MIDI to the Conductor Follower, which processed successive baton positions with tempo trackers that were implemented in the Max/MSP environment. There were three methods of tracking and predicting tempo. The first just used the time between the last detected beat and the beat previous to that to predict the time of the next beat. The second looked for six characteristic points in a beat curve, using zero crossings in velocity and acceleration of the baton, and thus allowed

to detect tempo changes between beats. The third method used the same characteristic points, but fed them into neural networks, which gave even better tempo control. The neural networks classified the conducting gestures and predicted the time of the next downbeat or half beat, with a downbeat being defined as a local minimum and a half beat being defined as a local maximum of the y coordinate of the movement of the tip of the baton. To train the neural networks, a conductor conducted along with a metronome for different tempi and the system adjusted the networks as necessary.

Guy Garnett continued to work on conducting systems and in 1999 he, Fernando Malvar-Ruiz and Fred Stoltzfus presented their *Virtual Conducting Practice Environment*, where they focused on determining appropriate aural and visual feedback on what student conductors are doing right or wrong and tried to build a system that can emphasize different aspects of conducting, depending on the skill level and learning goal of the student [GMRS99]. This system also used a Buchla Lightning as input device. It gave graphic representations of simple features, like the position of recognized beats in the beat plane, or more complex features, like whether a conducting style tended more towards staccato or more towards legato. It could play sounds every time it recognized a beat or it could play a simple melody that followed the student in tempo and loudness. But in the paper describing their system, the authors concluded that the system could replace neither a good teacher nor practice with live musicians.

In 2001, Guy Garnett, Mangesh Jonnalagadda, Ivan Elezovic, Timothy Johnson and Kevin Small published a paper about technological advances for their *Interactive Virtual Ensemble* [GJE+01]. In particular, they noted that the position information given by the Buchla Lightning baton was insufficient for their needs, and they abandoned it in favor of a sensor system called MotionStar, built by Ascension Technologies. In this sensor system, a pulsed magnetic field was picked up by magnetic sensors that measured their own position and orientation relative to the source of the field and transmitted the values wirelessly to a receiver that was connected via ethernet to the host computer. In the Interactive Virtual Ensemble, the sensors were used to track the baton position in three dimensions and to obtain pitch, yaw and roll of the baton, as well as hand and head motion of the conductor. The system followed a distributed processing model, where one computer processed all sensor input, deriving controller data from it, and another computer requested only the controller data it currently needed from the first computer. The system used neural networks for beat prediction and classification, and instead of sending out notes to MIDI synthesizers, it featured sound output based on sound analysis and resynthesis, controlled directly by conducting information.

### 2.2.8 The Ensemble Member and the Conducted Computer / Extraction of Conducting Gestures in 3D Space

In 1995, Forrest Tobey developed a software system that tracked tempo along all points of the path of a baton and allowed the conductor to take or release control of musical phrases [Tob95]. It also included a rehearsal module, so the conductor could train it to his gestures, and it used a Buchla Lightning baton as input device, which yielded two-dimensional position information. Forrest Tobey and Ichiro Fujinaga extended the

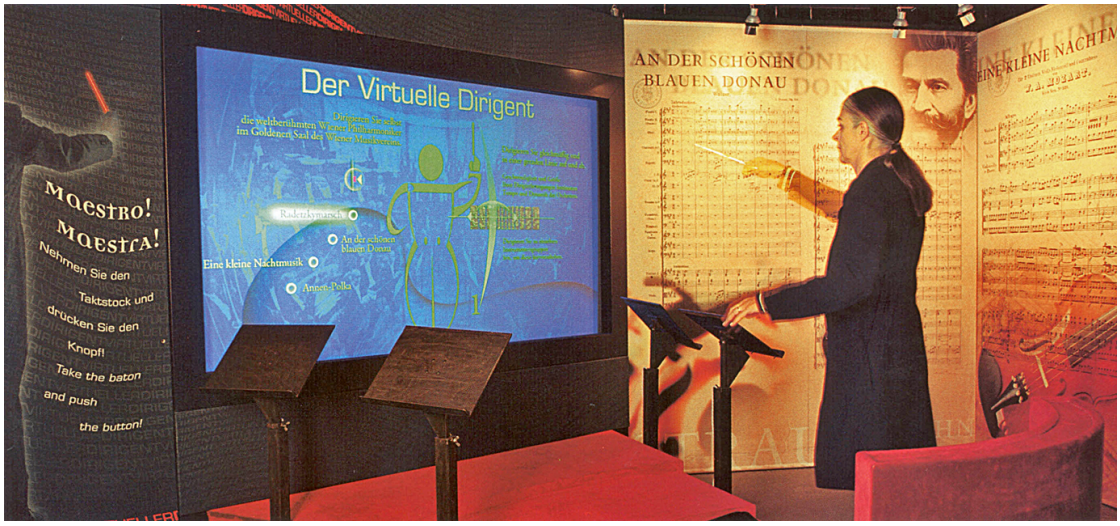


Figure 2.10: Der virtuelle Dirigent. Image courtesy of the Media Computing Group at the RWTH Aachen.

system in the year 1996 [TF96]. The extended version of the system included a second Buchla Lightning sensor. With two sensors, movement of the baton could be tracked in three dimensions. This extended system featured tempo control, dynamic control, beat pattern recognition, beat style recognition, accentuation control and timbral balance.

### 2.2.9 WorldBeat, Personal Orchestra and You're The Conductor

In 1996, Jan O. Borchers designed the *WorldBeat* system [Bor97, Bor01]. It was one of the exhibits shown in the Ars Electronica Center in Linz, Austria. It featured several modules showing visitors how computers could aid interacting with music in new ways, for example by trying to find the titles of songs visitors hummed into a microphone. Apart from the module using the microphone, all user input to the system came from a Buchla Lightning baton system. WorldBeat also included a module that let users conduct a piece of music, but the algorithm to track conducting movements was not developed originally for WorldBeat. Instead, the system reused components from Guy Garnett's Conductor Follower. These components were slightly modified to be usable by visitors who had no prior conducting experience.

In 2002, Jan Borchers, Wolfgang Samminger and Max Mühlhäuser finished the *Personal Orchestra* system [Sam02]. It was another museum exhibit, this time for the House of Music in Vienna, Austria. The system was developed under the name Personal Orchestra but the museum called the exhibit *Der virtuelle Dirigent* (i.e., Virtual Conductor), even though the orchestra was virtual and not the conductor. Visitors could interact with the exhibit by using a Buchla Lightning baton to first select one of four musical pieces and then to conduct the selected piece. An important focus in developing Personal Orchestra was to use recorded audio and video of the Vienna Philharmonic Orchestra for output, adjusting playback with the help of manually created beat files


Figure 2.11: You're The Conductor exhibit. Image taken from [LNB04].

containing the beat times of the musical pieces. Because no MIDI score was used, a conductor could only control the tempo and volume of playback as well as emphasize certain instrument groups. Conducting gesture recognition was very basic, a beat was detected each time the baton changed from going down to going up and users were advised to use a simple up and down conducting movement instead of more elaborate beat patterns. The computers used for the exhibit were not fast enough to change the tempo of the audio recording in real time in high quality. Therefore, the audio was preprocessed, several tracks with pitch-shifted versions of the original audio were created with a slow algorithm that kept the audio quality. Pitch-shifted instead of time-stretched alternative tracks were used because they all had the same length and could be placed alongside the video recording in a QuickTime movie file, ensuring synchronization of video and audio during playback. As slowed down playback of audio lowers pitch and sped up playback raises pitch, changing playback speed only required setting the correct frame rate for the video and activating the one audio track that featured a pitch shift that cancelled out with the playback speed, while muting all other audio tracks. Old and new active audio track were cross-faded to avoid noticeable audio glitches. Of course, there was only a limited number of audio tracks, so playback speed could only be adjusted at discrete intervals and a conductor could conduct faster or slower than the orchestra was able to follow. To avoid having to give an error message in such cases, thus ruining the user experience of the exhibit, video sequences were recorded that showed members of the orchestra complaining. If the conductor conducted outside the speed range of the orchestra for some time, he would be shown a complaint scene. Not only did the museum

visitors accept this kind of error message, they rather liked it, and a lot of people tried to conduct badly in order to see all of the complaint sequences.

Another museum exhibit was created in 2003 by Eric Lee, Teresa Marrin Nakra and Jan Borchers for the Children's Museum in Boston, USA [LNB04]. While still in development, it was nicknamed *Personal Orchestra 2*, but the finished system was called *You're The Conductor*. Like Personal Orchestra, it output recorded audio and video of a real orchestra. Faster computers enabled the system to time-stretch the audio in real time by using an improved phase vocoder algorithm, enabling the system to play back the recording at any wanted speed. Instead of employing a Buchla Lightning baton for input, a very rugged baton-like device was developed, which was mainly a light source and could stand heavy use. The light point was tracked with a camera, but no real conducting gesture recognition took place: any movement of the baton was translated into playback speed and volume, so that children of all ages could use the system. If a child moved the baton faster or slower, the orchestra sped up or slowed down, respectively, and if the child stopped moving the baton, the orchestra slowed to a halt.

### 2.2.10 Digital Baton, Conductor's Jacket and Gesture Construction

Teresa Marrin and Joseph Paradiso presented their *Digital Baton* in 1997 [MP97]. They had developed an input device similar to a baton, which included three acceleration sensors to measure movement of the baton and five pressure sensors to measure the pressure of each finger of the hand holding the baton as well as an infrared LED at the tip of the baton. A position-sensitive photodiode was placed behind a camera lens to track position and intensity of the infrared LED. Musical performance applications developed for the Digital Baton allowed the user to influence the volume of several tracks of a prerecorded MIDI score, to trigger playback of samples as well as placing the triggered samples in the stereo panorama and to switch instruments, re-orchestrating the piece. There was no tracking of conducting gestures and no beat information was derived from the input to control the tempo of the piece being played. However, Teresa Marrin's thesis paper on the Digital Baton did include a classification and analysis of conducting gestures [Mar96].

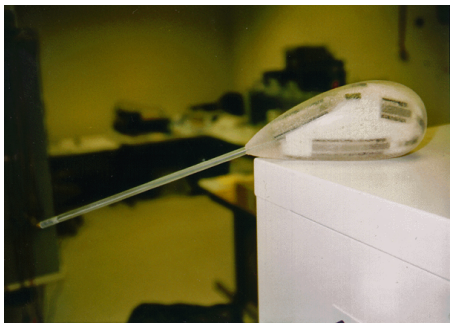

Figure 2.12: The Digital Baton. Image taken from web.media.mit.edu /~joep/TTT.BO /Baton2.gif

Teresa Marrin and Rosalind Picard developed the *Conductor's Jacket* in 1998 [MP98]. The system used a multitude of sensors built into a jacket to be worn by musicians to record physiological and motion data. The monitored physiological aspects were muscle tension of various muscles on each arm, heart rate, temperature, respiration and skin conductance. Motion data was acquired using an UltraTrak motion capture system from Polhemus Corporation. The jacket was connected to a computer running utilities written in National Instruments' LabVIEW package, collecting and analyzing the data.

This first version of the system did not use the acquired data to let users conduct a computer. After an in-depth analysis of the data sets collected with the first version, Teresa Marrin created a second version that did allow to conduct a prerecorded musical score, generating sound output by using MIDI synthesizers [Nak00]. She modified the jacket to include only the respiration sensor and the sensors for muscle tension, placing those at biceps, forearm and hand of each arm, as well as on the right shoulder. She also wrote the musical software *Gesture Construction* that detected beats based on maxima in muscle tension of left and right biceps. The software also detected holds, cutoffs and pauses, and enabled the conductor to control tempo, note and channel volumes, articulations and accents. It also allowed the conductor to choose pitch and number of voices, to pan instruments and change instrument balance, to morph timbres and do several other performance oriented things, moving beyond what a real orchestra is capable of. Gesture Construction was split in two parts running on separate machines. The part for input data acquisition and filtering was implemented in LabVIEW, while the performance part holding the musical functions and creating the output was written in C++.

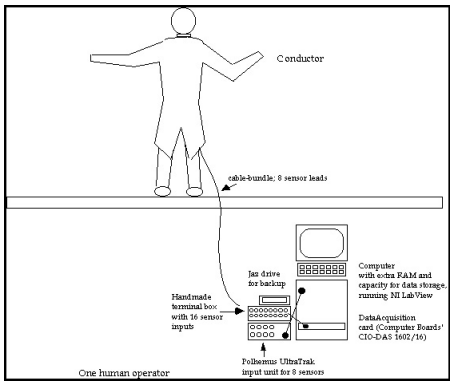


Figure 2.13: Data acquisition setup for the Conductor's Jacket. Image taken from [Nak00].

### 2.2.11 Multi-Modal Conducting Simulator

In 1998, Satoshi Usa and Yasunori Mochida built their *Multi-Modal Conducting Simulator* [UM98]. To recognize beat patterns conducted with the right hand, acceleration sensors were used, tracking movement of the baton in two dimensions. Hidden Markov Models detected the beats and fuzzy logic production rules determined if the detected beats were considered to be valid. To give cues to different instrument groups, an eye tracking camera detected which instrument group the conductor was looking at. A breathing sensor allowed to couple certain passages in the score to breathing patterns of the conductor. The beat patterns used by the conductor determined tempo and volume, and the system could also differentiate between staccato and legato beat patterns. The Multi-Modal Conducting Simulator generated sound output with MIDI synthesizers, taking the notes to be played from a prerecorded score that also contained special markers for cues and for breathing patterns.

### 2.2.12 Virtual Orchestra

Frederick Bianchi, Jeff Lazarus and David B. Smith founded the company Realtime Music Solutions in 1998, to commercialize their *Virtual Orchestra* system [Sol]. Virtual Orchestra could be conducted exactly like a real orchestra and it sounded very much like a real orchestra as well. Human technicians served as interface between conductor

Figure 2.14: Multi-Modal Conducting Simulator. Image taken from [Sam02].

and computer, interpreting the conducting gestures and instructing the computer accordingly in real time. The system used prerecorded pieces to control a synthesizer that played samples taken from real instruments. Musicians playing in real orchestras did not welcome Virtual Orchestra because they feared to be replaced by a computer. Later the system was renamed to *Sinfonia*, and a simpler variant called *OrchExtra* was created.

### 2.2.13 Conductor Following with Artificial Neural Networks

Tommi Ilmonen and Tapio Takala developed a system to track conducting gestures with neural networks, starting development in 1998 and finishing it in 1999 [Ilm98, IT99, Ilm99]. Their publications did not mention an official name for their system, maybe they did not name it because it was part of another system: it served to drive the virtual orchestra of DIVA, the Digital Interactive Virtual Acoustics project group [Ilm]. This virtual orchestra featured stylized 3D models of musicians of a band, whose motions were calculated from the notes of a MIDI score. Sound output of the virtual orchestra was generated from the same MIDI score, simply using a MIDI synthesizer for some of the instruments and synthesizing others on a computer using the physical modeling technique.

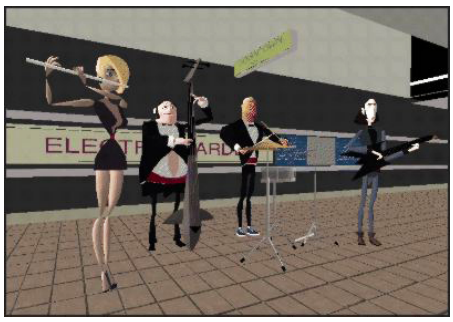

Figure 2.15: DIVA virtual orchestra. Image was taken from www.tml.hut.fi /Research/DIVA/past /imgs/band1.jpg

However, the conductor following system was in itself already very advanced. It used Ascension's MotionStar magnetic motion tracker as input device, with one sensor placed

at the conductor's left hand, another one at the right hand or alternatively mounted on the baton and one at the conductor's neck as reference point for the other two. Therefore, input data for the neural networks consisted of three-dimensional positions of the sensors at the conductor's hands relative to the position of the sensor at the conductor's neck. Ilmonen also tried using accelerometers placed on the baton instead, but he found that they were significantly less accurate than the magnetic sensors. The system used the neural networks to classify and predict beats, and they were even able to identify subdivided beats. It let conductors mainly control tempo and volume of the DIVA band, but the neural networks could also distinguish between staccato and legato beat patterns, making the band play accordingly.

Figure 2.16: DIVA conductor following system sensor placement. Image taken from [Ilm98].

### 2.2.14 Virtual Dance and Music

Jakub Segen, Aditi Majumder and Joshua Gluckman created their *Virtual Dance and Music* system in the year 2000 [SMG00]. The system had three main parts: a gesture recognition system, a music sequencer and a dance sequencer, with the gesture recognition system driving both sequencers to produce synchronized music and dance as output. The gesture recognition system used computer vision to extract beats from conducting gestures. Two synchronized cameras acquired a three-dimensional trajectory of the baton and beats were placed at the locally lowest points of the trajectory. To reduce latency, the gesture recognition system predicted beats if possible, by using a polynomial predictor that indicated the quality of its predictions. If prediction quality was poor, beats were detected instead of predicted, resulting in higher latency. The music sequencer played music from a MIDI file, adjusting the tempo to the time intervals between beats detected by the gesture recognition system. If the music sequencer had already played all notes corresponding to the current beat, it just waited until the user conducted the next beat, and if the user conducted the next beat before the music sequencer finished playing the notes corresponding to the last beat, it increased the tempo slightly more than indicated in order to catch up with the conductor. The dance sequencer created video output of human avatars dancing to the music, according to the beats given by the conductor and constrained by a model of possible human motions and laws like gravity, to prevent unnatural and impossible dance motions.

### 2.2.15 Conducting Audio Files via Computer Vision

Declan Murphy, Tue Haste Andersen and Kristoffer Jensen developed a conducting gesture recognition system in 2003 [Mur03, MAJ03]. It could work with one or two cameras as input sources. One camera was always taking a front view of the conductor and if a second camera was present, this second camera was taking a side view of the conductor. Computer vision techniques were then used to extract position and velocity of the tip of the baton or of the conductor's right hand, if he conducted without a baton.

Several standard beat patterns, encoded as template functions of an even tempo, were compared to the position and velocity data from the tracker to follow the conductor's progress in the beat pattern, allowing users to control tempo and dynamics of the piece. The conducting gesture recognition part was implemented in the EyesWeb software environment, creating MIDI output for the audio system. The audio system did not use MIDI synthesizers to generate sound but instead used a phase vocoder to change the tempo of an audio file. Beat positions in the audio file were calculated automatically, no score file or manually created beat information was needed. Tempo coupling between conducting and audio playback could be done in two modes, with a mode for low latency input trying to keep the audio in direct sync with the beats conducted by the user and a second mode for high latency input trying to catch up with the conductor at the estimated time of the beat following the last detected beat.

Figure 2.17: Declan Murphy using his conducting system. Image taken from [Kol04].

### 2.2.16 Conducting Gesture Recognition, Analysis and Performance System

In 2004, Paul Kolesnik created his *Conducting Gesture Recognition, Analysis and Performance System* [Kol04]. The system consisted of two parts running on separate machines. A Windows PC executed the input part which recorded a front and side view of the conductor with two USB cameras. The input part used components of the EyesWeb software to find both hands of the conductor and extract their two-dimensional position in the camera view. To aid this process, the conductor wore a colored glove on at least one hand. Position data of both hands was then passed to the second part, which was implemented in the Max/MSP environment and running on a Macintosh computer. This second part derived beat and amplitude information from the conducting gestures of the right hand and expressive information from the gestures of the left hand. Paul Kolesnik developed a package of Hidden Markov Model tools for Max/MSP that were used to process and classify the gestures. HMM objects pairs[6] were created and trained

[6] One HMM object for each camera view.

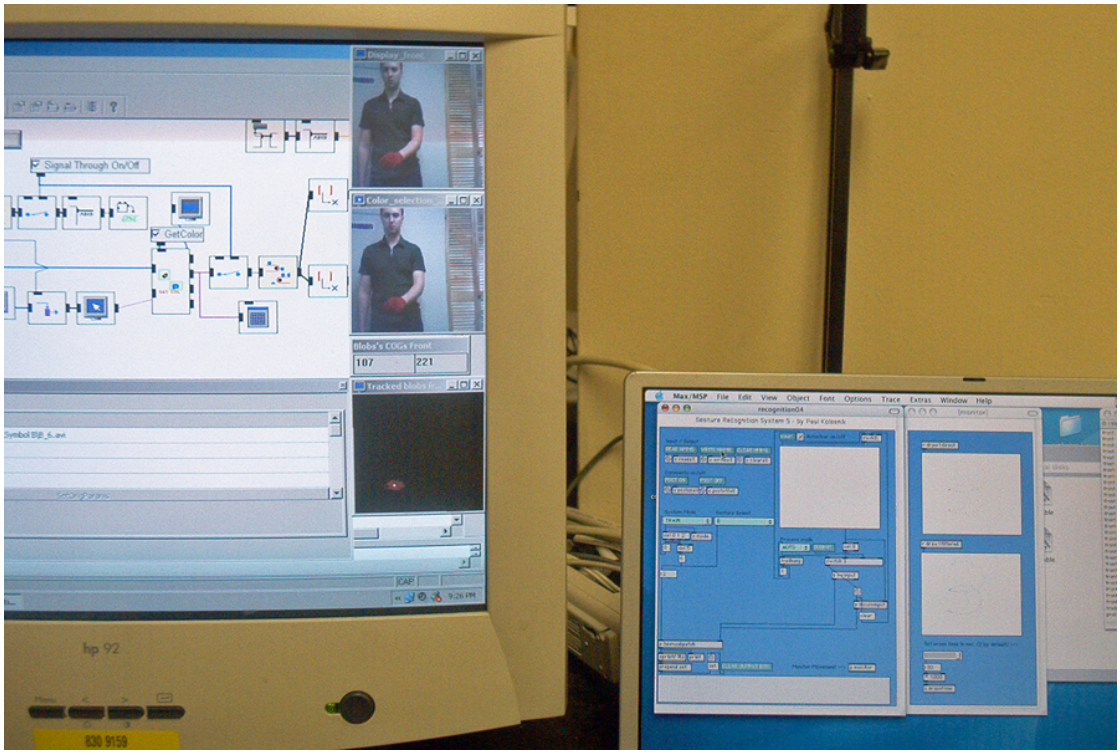


Figure 2.18: The parts of Paul Kolesnik's system side by side. Image taken from [Kol04].

for each gesture that the system should accept as input. In performance mode, all HMM objects processed the input data and for each hand the system chose the HMM object pair that was considered most likely to correspond to the current conducting gesture, deriving the information needed for controlling system output from the chosen HMM object pair. Output of the system was either audio only or audio and video combined, with the audio output being created by time-stretching an audio file using Max/MSP components like a phase vocoder. To be able to use an audio file for output, a file with the beat times of the audio file had to be created manually, using a component Paul Kolesnik wrote for that task. Paul Kolesnik encourages reuse of his work, he made the HMM package available for free distribution[7].

[7]The HMM package can be downloaded from http://www.music.mcgill.ca/~pkoles/download.html

## 2.3 Comparison of Computer-Based Conducting Systems

It is hard to compare the presented computer-based conducting systems because of the very different approaches that were taken in their development.

There are a lot of parameters a trained conductor can influence when conducting a real orchestra. He can for example indicate when notes will be played and how long they should be, holding them or cutting them off. He can tell the musicians how loud notes should be played, how notes should be accentuated and what form transitions between subsequent notes should take. He can influence how notes are grouped or at least perceived as groups. He has control over the whole orchestra as well as the different instrument groups and can dictate when and how they enter the musical piece. He can emphasize an individual instrument group and control this group directly for some time, while the rest of the orchestra serves as background. Or he can let his attention jump from instrument group to instrument group, issuing commands to each group, fine-tuning their balance and interplay. Supporting all those parameters in a computer-based system, and supporting them well, is very hard indeed, and different systems pick different subsets of those parameters, with some systems featuring only the basics like tempo and volume and others going beyond that in letting the conductor shape the notes in various degrees or giving him control over instrument groups. How the parameters present in the respective systems can be influenced by the conductor also differs from system to system. There are systems that just let the conductor enter a value for each parameter he wants to change. Others derive the parameter values from conducting gestures, hand gestures or eye contact. Virtual Orchestra even uses humans as transducers for conducting input, in order to capture nuances a machine would miss.

What parameters are supported is also influenced by the intended user group. Some systems are targeted towards people with no conducting experience whatsoever, some address people who are interested in performing music but do not want to learn an instrument or do not want to have to control all aspects of the sound they generate, as they would have to when using a musical instrument. Some are aimed at trained conductors and some are aimed at students still learning how to conduct.

The systems also use very different input hardware, ranging from simple keyboards and switches over cameras and data gloves to all sorts of sophisticated sensors, each having their own characteristics, with latency and accuracy being of special importance. Input data can be video images of the conductor, of body parts like his hands and his eyes, or of the baton he is wielding. Several enhanced batons are employed in the systems, some acting as a light source whose position can be tracked, again with different kinds of sensors to locate the point of light. Other batons provide acceleration values, sometimes only in the form of triggers. Even radio waves are used to locate the baton, or magnetic fields serve to measure position and orientation of the baton and/or the conductor's hands. Sensor technology from the medical field has been used to monitor body signals of the conductor. Very different ways of processing the input were tried out, starting with functions to de-noise the signals and rather simple algorithms to extract triggers from input data and ending with very advanced algorithms like neural networks and hidden markov models. Although the software of the systems varies wildly, there

have also been cases of reuse of existing components, or components have been built in order to be reused, facilitated by utilizing frameworks such as Max/MSP.

The output of the systems takes several forms as well. Some play back recorded audio and video, some output MIDI data and rely on MIDI synthesizers to generate their sound. Some incorporate advanced synthesis algorithms like physical modeling of instruments into the system in order to be able to control very precisely how the sound of the orchestra is generated. The performance based systems do even synthesize graphics or animations based on the conducting input.

A very important point is that different systems use different definitions of conducting gestures. Some see conducting gestures just as a sequence of beat triggers. Others define and measure conducting gestures as the movement of the conductor's baton or hand in one, two or three dimensions, often reducing the properties of the baton to the position or acceleration of its tip and placing the beats at local minima of its vertical movement. Some of the systems defining conducting gestures as movement of the baton take the form of the trajectory into account, i.e., there are explicit models of beat patterns or components that can be trained to recognize beat patterns. In a completely different approach, the Conductor's Jacket regards conducting gestures as patterns of physiological signals. Several systems detect special gestures that add to the basic conducting gestures and model those as posture and orientation of the left hand or direction in which the conductor is looking.

For perfect control of a conductor over a real orchestra, the musicians of the orchestra and the conductor must know each other and rehearse a musical piece before performing it. Only some of the presented systems take this aspect of conducting into account.

# 3 The Context that Led to the Conga Framework

The Media Computing Group at RWTH Aachen, the Rheinisch-Westfälische Technische Hochschule in Aachen, Germany, was established in the year 2003. With Jan Borchers as head of the group and Eric Lee as research assistant, computer-based conducting systems belong to its history. When the group pondered development of successor systems to Personal Orchestra and You're The Conductor, the question arose, how the conducting gesture recognition parts of those systems could be reused and improved. It turned out that, because the gesture recognition parts had been held simple, they had been implemented in a way that made extracting them and encapsulating them for further reuse just as hard as re-implementing them from scratch in a new system. In addition, substantial improvements in the way they tracked conducting gestures would have required a redesign anyway. In consequence, it was decided to reuse neither the gesture recognition code of Personal Orchestra nor the one of You're The Conductor in successor systems. To avoid running into the same problem in the future, the group started looking for components that would allow to create an encapsulated gesture recognition part that is isolated from the rest of the system and that would also allow to adapt such a part to changed requirements. Moreover, these components should accommodate the use of very different input hardware, as not even Personal Orchestra and You're The Conductor used the same input devices. They should also enable building conducting gesture trackers that do not depend on the way sound is generated from conducting input, since different ways of doing so have been implemented in the past and sound synthesis is still a developing field.

Frameworks like LabVIEW, EyesWeb and Max/MSP meet these demands. Their components are tried and tested, each framework has its own user community and the commercial frameworks feature additional support by the companies selling them. The frameworks have been used successfully in several computer-based conducting systems. There are even modules like neural networks or hidden markov models, developed with these frameworks for such systems, that are available for reuse. But there are drawbacks to using these frameworks as well. The modules for advanced tracking of conducting gestures have been developed precisely because there are no built-in components specializing on conducting. The graphical programming environments are very comfortable, but programmers are forced to use them because the modules of the frameworks cannot be used easily outside the frameworks. If one wants to combine these modules with code from other sources, one is either forced to package that code into external modules for the framework and use the framework's programming tools to develop the overall application, or one has to create two applications, using the framework's runtime environment

and an operating system process running in parallel. In the latter case, the application borders will be dictated by the types of the parts one wants to combine. Furthermore, one has to insert glue code into both applications so they can communicate with each other, adding to the complexity of the system. Another drawback might be the way the frameworks handle the flow of data and how data is processed. These are fixed because the frameworks have been used by many people to develop a lot of different applications — changing the framework's core behavior would break those applications. So a programmer using the frameworks has to adapt his application logic to fit the framework.

In the end, the group decided that it would be better to create a new framework that specializes on conducting gesture tracking and particularly supports a better set of operators for conducting gestures than the existing frameworks. It should also allow to create processing units that can be mixed with existing code that was developed with standard development tools. This framework was to be called *conga*, short for "CONducting Gesture Analysis framework", and it would first be used in the *Personal Orchestra 3* project of the Media Computing Group. Objective of the project was to create an interactive conducting system for The Betty Brinn Children's Museum[1] in Milwaukee, USA, featuring an adaptive gesture recognition that could act like You're The Conductor, deriving tempo and volume from any baton movement, but that could also track a standard beat pattern giving the conductor finer control than in Personal Orchestra. Personal Orchestra 3 would also feature audio output using a phase vocoder that was more advanced than the one that had been used in You're The Conductor. Thorsten Karrer was working on this part of the project for his diploma thesis[Kar05][2].

Creating a new framework meant to forego the advanced conducting gesture recognition modules available for reuse that have been built with existing frameworks. But this was deemed acceptable, because these modules use neural networks or hidden markov models and thus rely on being trained to certain beat patterns. The future conducting systems planned at the Media Computing Group will probably be interactive music exhibits. This means they will be used by each user only for a short amount of time and probably only for a few times or even only once. That makes it impractical to train the systems to each user. But if one uses general training sets to train the gesture recognition modules, one cannot be sure if the modules really learned the essential properties of the gestures. In addition, it is not possible to change the way the module tracks a gesture without retraining it.

[1] A museum installation will not be maintained and operated by computer experts. Therefore, it is desirable to have a system that only needs a single computer to run on. As the target platform of Personal Orchestra 3 is an Apple computer with MacOS X, this would have ruled out EyesWeb anyway.

[2] Both the Personal Orchestra 3 project and Thorsten Karrer's diploma thesis were still in progress at the time this thesis paper was written, so the final title and publishing date of his diploma thesis might differ from the ones given in [Kar05].

# 4 The Conga Framework

This chapter describes the fundamental characteristics of the conga framework, particularly the underlying model of what a conducting gesture is plus the important concepts and structures of the framework and its components. It also mentions some of the problems in developing conga and the impact they had on the design of the framework.

## 4.1 Model of Conducting Gestures

In her PhD thesis about the Conductor's Jacket ([Nak00]), Teresa Marrin shows that conducting is an activity that involves the conductor's whole body. But the actual conducting gestures mainly rely on the conductor's hands. According to Paul Kolesnik, who focused on recognition of expressive gestures in his Master's thesis ([Kol04]), expressive gestures in conducting can be performed with either hand, but time-beating gestures are almost always carried out using the right hand. Max Rudolf is even more strict, in his book *The Grammar of Conducting: A Comprehensive Guide to Baton Technique and Interpretation* ([Rud95]) he argues that all important conducting information *can* and *should* be conveyed with gestures of the right hand, citing the conductor Richard Strauss: "The left hand and *both arms* are dispensable, a good wrist is sufficient." The book describes conducting gestures that can be performed with the right hand, both with or without a baton, and these gestures cover musical expression as well as time-beating. It explains the shape of the gestures with illustrations that show the path followed by the tip of the baton when performing a certain gesture.

The design of the conga framework is based on the model of conducting gestures given in Max Rudolf's book, i.e., conga models conducting gestures as two-dimensional trajectories of a point, with the point corresponding to the tip of the baton, or the conductor's right hand, if he is conducting without baton. In his book, Max Rudolf depicts the shapes of the trajectories of several fundamental beat patterns, including the placement of the beats. He also supplies information whether or not the baton or right hand should stop on the beats and whether movement should be deliberate and controlled or rather very quick. Figure 4.1 is an example of various beat patterns for conducting gestures in the graphic notation of Max Rudolf. Properties shared by different beat patterns express themselves in similarities in the trajectories of those patterns, for example both neutral-legato pattern trajectories in said figure look similar, although one is a 3-beat pattern trajectory and the other is a 4-beat pattern trajectory. The numbers in the figure are placed at the locations of the beats and give the order in which the beats occur when the beat pattern is executed by the conductor. Consequently, each beat marks a point in time as well as a point in the trajectory of its beat pattern.

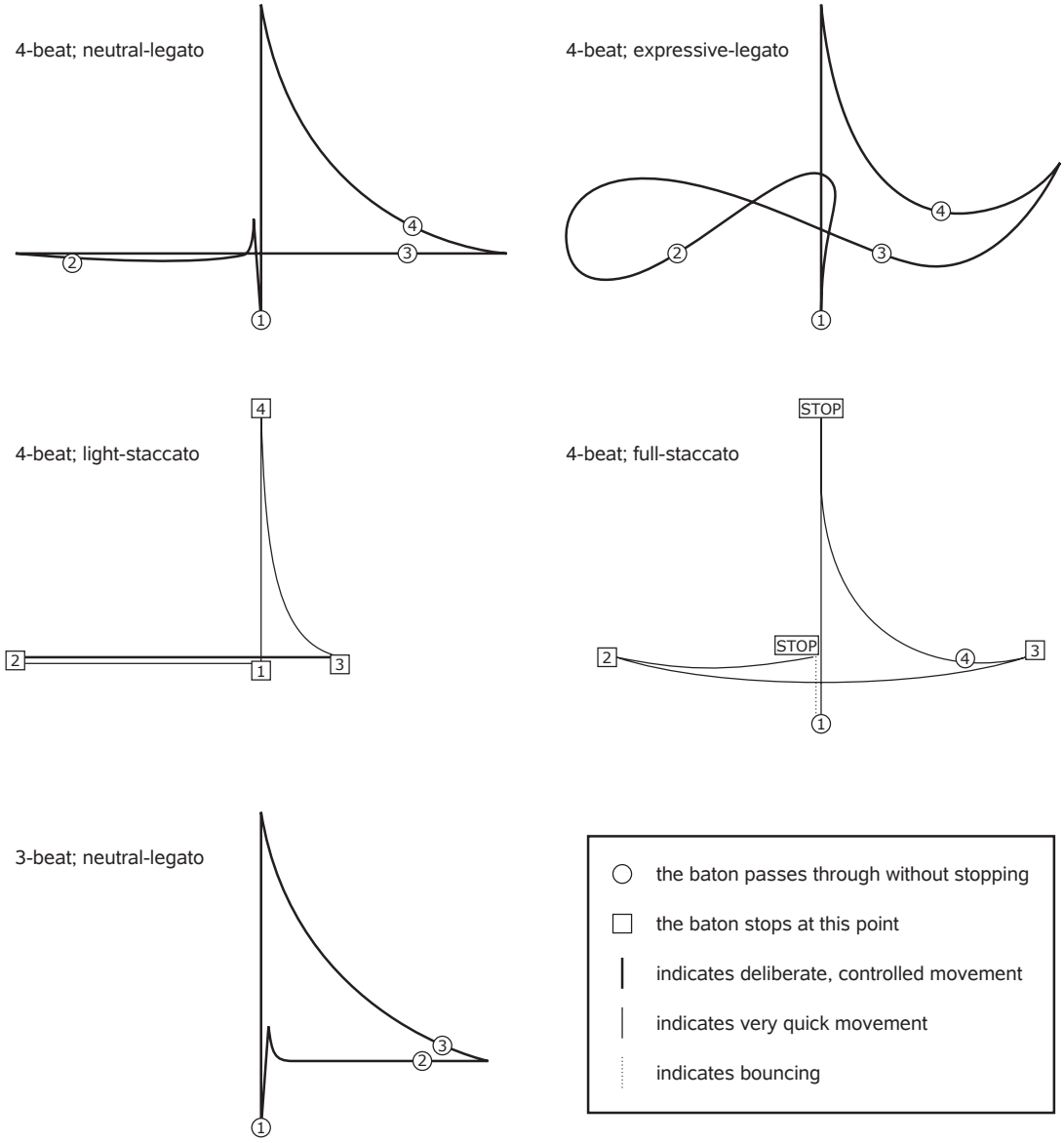


Figure 4.1: Some examples of Max Rudolf's beat patterns for conducting gestures. Drawings after [Rud95].

Conducting an orchestra with a certain beat pattern results in the same pattern being repeated over and over again. As long as the conductor sticks to the same beat pattern, he is in essence cycling through this pattern, with the beats marking distinct points in the cycle and the first beat in the pattern marking the boundary between subsequent cycles. Each full beat pattern drawn in The Grammar of Conducting represents one complete cycle in executing the beat pattern, showing the shape of the trajectory corresponding to the beat pattern's conducting gesture. From the shape of this trajectory follow certain features, such as the speed of the tip of the baton or the direction of its movement and so on. Outstanding features allow to identify a certain beat pattern, e.g., the overall shapes of the neutral-legato and full-staccato 4-beat patterns in Figure 4.1 might look very similar, but the stops featuring in the full-staccato pattern's trajectory allow to tell the patterns apart. Repetitive occurrences of distinct features make it possible to track the execution of a beat pattern, using those and other features to derive conducting information like tempo, volume, targeted instrument group, and so forth.

The conga framework provides objects that can be interconnected to process and analyze input data corresponding to points of the trajectory of the conductor's baton or right hand sampled at regular time intervals. It also provides the means to build feature detectors and to model the gesture cycles of beat patterns.

## 4.2 Choice of Platform and Implementation Language

The Media Computing Group uses mostly Apple Macintosh computers running the Mac OS X operating system, and a lot of existing code used and/or developed by the group has been implemented either in C++ or in Objective-C. Apple's development tools allow to mix Objective-C and C++ code, and the more modern parts of Mac OS X are mostly based on Objective-C, including the *Cocoa* APIs providing access to much of the functionality of Mac OS X. Because of this, conga was implemented as an Objective-C framework for Mac OS X. One additional advantage to Objective-C frameworks in Mac OS X is that they do not have to be linked statically. An application can load them at startup, making it easy to give applications depending on such frameworks access to improved versions, without having to recompile the applications.

## 4.3 Early Approaches and Their Problems

The conga framework was developed with Apple's integrated development environment *XCode*. Most of the testing and debugging work was done on an Apple PowerMacintosh workstation with 512 MB of RAM and dual G5 processors running at 2 GHz, using a Buchla Lightning II[1] infrared baton system as main input hardware.

[1]The Buchla Lightning II system consists of two batons that emit infrared light, a sensor unit and a control unit that processes the sensor input, acts as MIDI interface and also contains a General MIDI synthesizer. It is manufactured and sold by the company Buchla and Associates. Their website can be found at http://www.buchla.com with information about the Lightning II at http://www.buchla.com/lightning/index.html

During the early approaches to create the framework, some rather time-consuming mistakes were made by the author. For example, one mistake was to test the very first attempts to track conducting gestures by using a mouse and a Wacom graphics tablet as input devices. Only some time later was the Buchla Lightning baton employed as input hardware instead, with the result that components of the gesture tracker that had seemed to work ceased functioning properly. This could have been avoided if the input device with the most problematic characteristics had been identified first and then used for most of the testing right from the start.

Another mistake in the beginning phases of the framework was to develop framework components and code to track conducting gestures depending on the framework in parallel. The problem of that approach was that changes to the framework's structure and way of processing data required changing the tracking code as well. But changes were frequent because the underlying model of what constitutes a conducting gesture kept changing, as did the way the framework would handle conducting gesture analysis — the definitions given in Section 4.1 are only the end result. For instance, an early version of the framework focused on detecting the presence of certain features from position, velocity and acceleration input, but did not provide the means to assemble new feature detectors from existing ones. Code depending on this version of the framework had to be reworked substantially every time the framework changed. After that, another approach was taken: First a simple tracker for the 4-beat neutral legato pattern was built by iteratively adding improvements to the code of one central function. When the tracker was working reasonably, this central function was decomposed into its building blocks and from then on the next version of the conga framework was developed from scratch, by creating components corresponding to said building blocks.

Yet another unfortunate early decision was to use short sounds to indicate detected features. A better decision would have been to visualize detected features and the input data for the feature detectors. Because progress was a slower than expected, visualization tools were implemented later on and did provide valuable insight. It turned out that velocity and acceleration values calculated from position data delivered by the Buchla system were varying wildly because the Buchla system did not deliver data at regular time intervals, with noise being exaggerated by very short time intervals. Resampling the input data at regular time intervals reduced the noise considerably. If visualization had been used from the start, important characteristics of the input data would have been apparent much earlier.[2]

Unnecessary complexity was a problem that arose again and again. Several times in the development process it turned out that a structure or way of processing that had been added to provide certain functionality could be merged with an older counterpart or even discarded completely by changing the older counterpart slightly. Up to that point, creation and maintenance of the added part consumed time and effort, and depending source code is of course more complex when it uses two different concepts instead of

[2]Incidentally, the first visualization tools worked offline. Later a version was created that ran in parallel to the tracking code, based on Apple's Cocoa objects for drawing lines. But these Objective-C objects are optimized for printing and not for fast display on a monitor, so visualization choked off tracking until both were decoupled. It would have been better to base the visualization on OpenGL instead.

one unified concept. For instance, as opposed to the final approach described in Subsection 4.4.1, some earlier versions of the framework distinguished between several types of data but processed these types in similar ways. In effect, there were several separate code structures that did basically the same processing. Another example of unnecessary complexity were the predecessors of the specialized finite state machine (FSM) and the corresponding states presented in Section 4.6. These predecessors were based on feature detection events being generated outside the FSM and then being passed to the FSM to be analyzed by the FSM's states. Removing the events altogether resulted in an FSM and states that were less complex and more flexible. Analyzing how existing structures and ways of processing could be improved to support new functionality before creating new ones would have saved a lot of effort, but it is unlikely that this would have been feasible in every case. Sometimes one has to work out a concept first in order to see the similarities to established concepts. But even then comparing each new concept with the established ones as soon as it had been fleshed out it would have saved time and effort later on.

Not a mistake but likewise time-consuming was the fact that by designing conga from scratch, as opposed to basing it on one of the frameworks introduced in Section 2.1, some functionality that would already have been present in an existing framework had to be implemented and tested. This could have been avoided by designing the important parts of conga as a set of modules extending an existing framework. But that would have meant to accept the drawbacks described in Chapter 3.

Because of these circumstances, conga was developed in several iterations. The components of the first framework attempts were disposed of completely and the following attempts evolved with concepts being added and refined, occasionally being merged with other concepts and sometimes just being discarded. Even after the fundamental concepts of the framework had stabilized, components were still added and improved iteratively. As a result, conga has a grown structure and source code and its design is not as clear and elegant as the author had originally hoped for.

## 4.4 Basic Processing Model

The final version of the conga framework has not only been influenced by the experiences made in the context of the earlier approaches. Its basic processing model also incorporates some ideas already present in LabVIEW, EyesWeb and Max/MSP. These existing frameworks feature processing units that can be connected to form a graph with data flowing along its edges and being processed in its nodes. Similar processing units are part of conga. In fact, conga consists mostly of components that are either processing units or used in processing units. Those units are meant to be connected into directed acyclic graphs and are called conga nodes accordingly. As conga graphs are created in source code and not in a graphical programming environment, objects called conga ports, representing the edges of the graph, are used to connect the nodes. Each processing node supplies one or more port objects that can be used for outgoing connections and accepts a certain number of port objects as connections going into the node. The

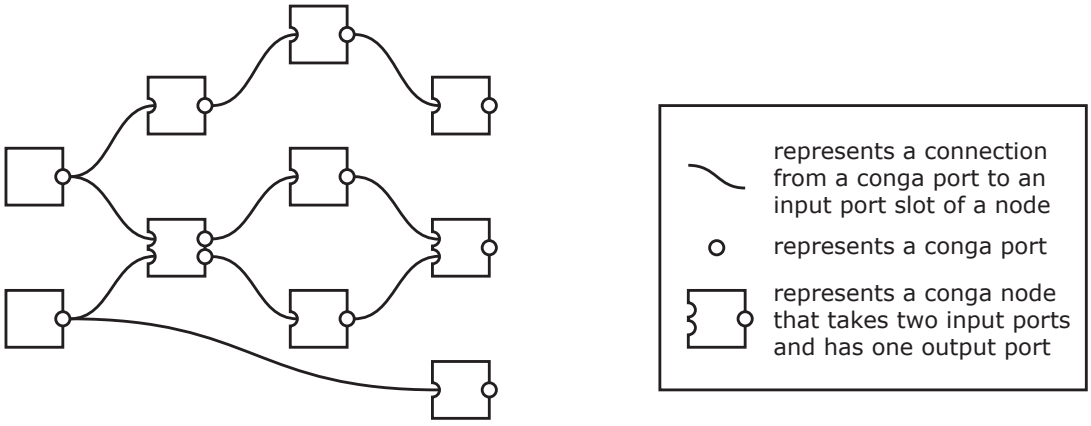


Figure 4.2: Visualization of the structure of a conga graph. Evaluation of the graph starts at the output ports of the nodes on the right end of the graph, and data flows from its left end along the connections to the right end.

input port objects a node takes have to be output port objects of other nodes — in this model a port that does not belong to a node would represent a connection from nowhere and thus does not make sense. Figure 4.2 visualizes the structure of a conga graph.

An outgoing port of one node can be used as input port for an arbitrary number of other nodes and not all outgoing ports of a node have to be used as input ports by other nodes. To accommodate this fact, data is only transported through the graph upon request from subsequent nodes. I.e., to evaluate a conga graph, input data is first supplied to the entrance nodes of the graph and then the outgoing ports of the exit nodes of the graph are asked to present their current values. The ports ask the nodes they belong to to calculate their values. To do so, the nodes request the current values of their input ports. The pattern repeats until the request reaches the nodes that have been supplied with input data. These can calculate a result and pass it on, so data travels through the graph in the reverse direction of the requests for it. In order to avoid repeated calculations to evaluate the same node, each request is marked with a timestamp. All nodes cache the result values for the evaluation request with the most recent timestamp and only recalculate their results if a newer timestamp is used by the next request. But due to the caching mechanism, *all* exit nodes of a conga graph have to be queried for results *every* time the graph is evaluated, otherwise there might be gaps in the sequence of data values that reach a given node. Because data flows only upon request, each node in the graph must get at least one request for each timestamp or else later requests with more recent timestamps might produce wrong results.

A simple conga graph will serve to demonstrate this. The graph calculates $5 + \frac{x}{y}$ as well as $3\frac{x}{y}$, with x and y standing for the values of two input signals. Figure 4.3 shows how this graph might be created, starting with the nodes that hold the input values and then creating and connecting all other nodes in steps that ensure that a given node is created after all the nodes it depends on.

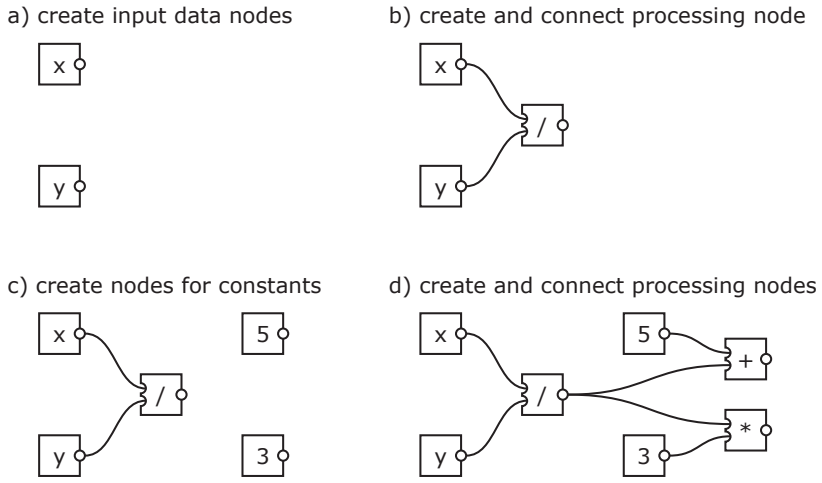


Figure 4.3: Creation of a simple conga graph. The nodes x and y are of the same type as the nodes 5 and 3, they are simply marked with characters to clarify that their values will be changed between successive graph evaluations.

Figure 4.4 walks through the steps of evaluating this graph:

a) The graph has been created with an initial timestamp of 0 and supplied with the current values of the input signal, x has been set to 4 and y has been set to 2.

b) The evaluation is started by requesting the value of $5 + \frac{x}{y}$ for timestamp 1.

c) The corresponding node has not been evaluated for this timestamp yet and in response first requests the value of the node with constant value 5, with no further requests being issued by that node.

d) The node calculating $\frac{x}{y}$ is asked next to yield its value for timestamp 1.

e) It requests the value of x, because timestamp 1 is more recent than timestamp 0.

f) It also requests the value of y for timestamp 1.

g) With those values, $\frac{x}{y}$ can be resolved to 2 and $5 + \frac{x}{y}$ can be resolved to 7, which the corresponding nodes cache alongside the timestamp 1. Then begins the evaluation of the graph's second output by requesting the value of $3\frac{x}{y}$.

h) The associated node has no cached value for timestamp 1 and so requests for this timestamp the value of the node with constant value 3.

i) Afterwards it requests the value of $\frac{x}{y}$.

j) The corresponding node already has a value for timestamp 1, so $3\frac{x}{y}$ can be resolved to 6 and cached in the associated node. In addition, the graph gets prepared for the next round of evaluation at timestamp 2, with new values set for x and y.

Section 5.1 has a short piece of Objective-C source code performing creation and evaluation of this graph, according to Figures 4.3 and 4.4.

The aforementioned properties of a conga graph, demonstrated by the given graph example, explain why conga processing units have to form directed acyclic graphs. The graphs have to be directed in order to allow data and requests for data to flow through them. They have to be acyclic because a cycle in a graph would result in a processing unit to request data from itself recursively, without an end condition for the recursion: an endless loop is born and the request is never answered.

### 4.4.1 Types of Nodes, Ports and Processed Data

conga needs basically three types of nodes. The first type just holds a data value and presents this value when queried for its current value. This type takes no input ports and can consequently be used for entry nodes of a conga graph, because requests for data reaching this type of node travel no further. The second type simply processes data presented by its input ports and passes on the results via its output ports. The third type is used to analyze data, to detect if a certain feature is present in its input data at a certain time or not and maybe to derive some values corresponding to the feature, if it is present.

Ignoring the node type that simply holds data as exceptional case, different nodes of even the same type may still operate on data of different dimensions. As conga is built on the assumption that conducting gestures have two-dimensional shapes, there are a lot of conga nodes that take two-dimensional input. But there are other nodes that take one-dimensional input and some nodes can even take an arbitrary number of input ports. The dimension of the output data of a node does not have to be, and in a lot of cases is not, the same as the dimension of its input data.

In order to be able to interconnect all types of nodes, all ports have the same interface and there is only one type of data that is passed through the ports. The data takes the form of two simple numbers. One is the value of the port and the other is an estimated time when said value originated. Because different input sources have different latencies and processing may, and indeed most of the time will, add further delays, passing the estimated origination time of a value along and modifying it as needed means that code using conga graphs can take accumulated latency into account, which for example is very useful for determining the actual time when a certain beat was given by the conductor. Some nodes use the timestamps of evaluation requests to modify the estimated origination time. Therefore, those timestamps should have a meaningful relationship to the origination time values given to the starting nodes of a conga graph.

Going back to the topic of node types and input/output data dimensions, the unified interface for ports means that the data dimension is reflected in the number of ports. A node taking two-dimensional input and giving one-dimensional output takes two input ports and has one output port, respectively. The unified port interface also means that a feature detecting node has to use numbers to indicate boolean values. For that purpose, conga incorporates the same trick that, for example, is used in the C programming language. A number equal to zero represents the boolean value *false* and any non-zero

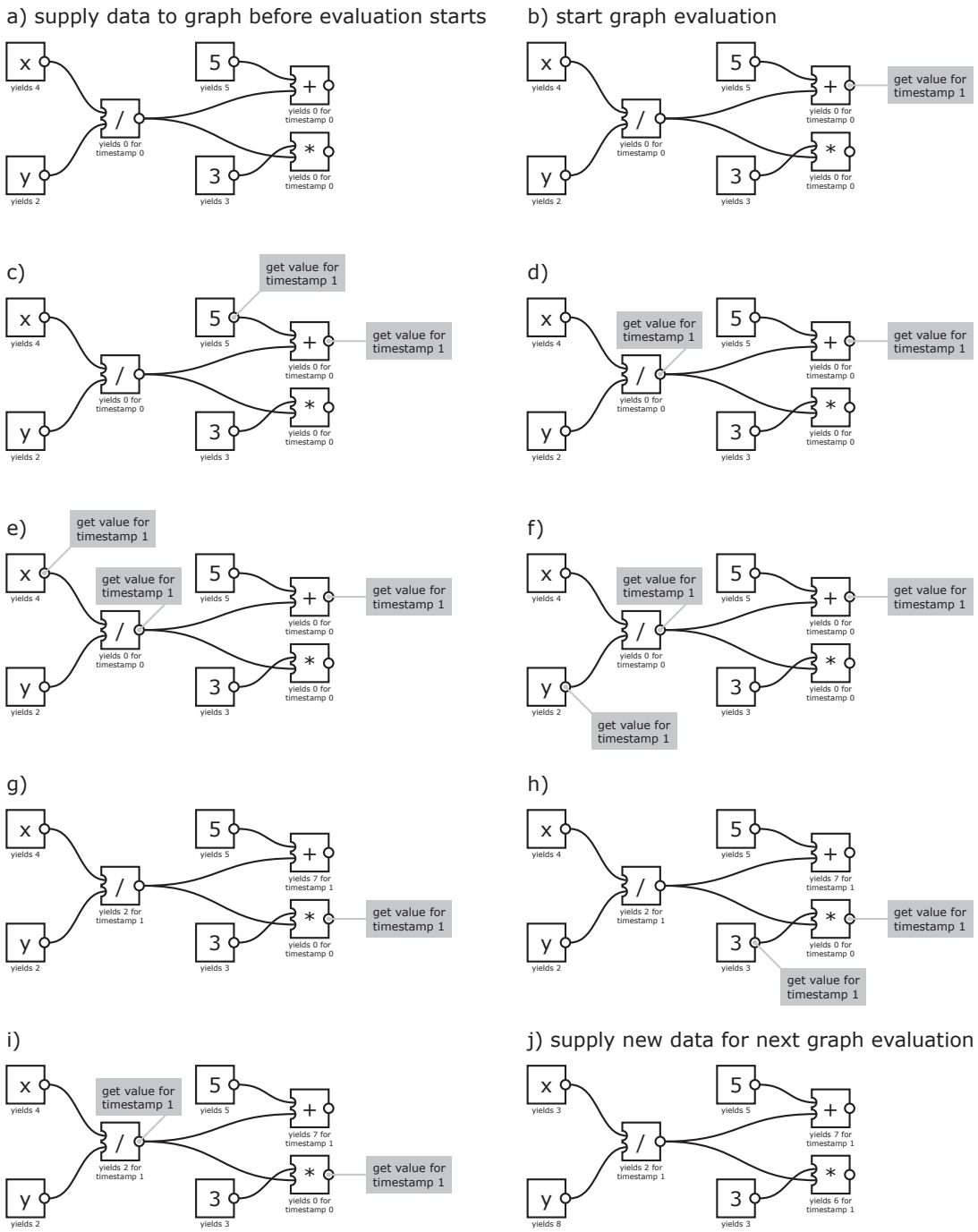


Figure 4.4: Sample evaluation of the graph from Figure 4.3 for the first time immediately after the graph was created.

number represents the boolean value *true*. Just like in C this presents the opportunity to use a node that was not actually intended to be a feature detector in place of a genuine feature detector, if it is convenient, and just like in C this poses the danger of not noticing that something is used as a boolean value although it is not appropriate to do so. However, there is an additional benefit to this arrangement: conga can use a numeric value to not only indicate if a feature is present, but also to indicate to what extent it is present.

To simplify matters, conga has been implemented with a unified basic interface for nodes as well, so that this basic interface depends neither on the number of input ports a node takes nor on the number of output ports it has. Since the behavior of a lot of conga nodes can be configured, those nodes extend the basic node interface with whatever methods they need to be set up properly.

In order to encapsulate a complete or a partial conga graph into a new conga node, this new node at least has to implement the methods in the basic interface for nodes. If the encapsulated graph does not need to be configurable and does not depend on activity outside conga (like, say, the input nodes being set to the current input values prior each graph evaluation), then all the new node has to do is to create this graph correctly, pass provided input ports to the right nodes of the graph and present the graph's exit ports as the new node's output ports. If the encapsulated graph does have to be configurable or or does have to be accessible outside conga's basic processing model, the new node also has to extend the basic node interface in order to provide the needed configuration and access methods.

### 4.4.2 Implementation-Specific Processing and Initialization Issues

Ports and nodes in conga are implemented as Objective-C classes. To form a conga graph, objects of these classes are instantiated, and for evaluation requests, messages are sent to the appropriate objects. Objective-C determines at runtime if a given object implements a method corresponding to a given message. A message passed to an object thus results in a method lookup and a subsequent method call. This process takes more time than a simple function call and because a conga graph might consist of a lot of node and port objects, the overhead of this process could impact noticeably the time it takes to evaluate the graph. To avoid this, conga nodes and ports have been implemented so that they construct function pointers to the important methods of their input ports and host nodes whenever those are being set. This makes it possible to circumvent the method lookup and speed up the evaluation of a conga graph. It also makes the source code of the nodes and ports less elegant and more prone to bugs.

All conga nodes share the same basic interface which is defined as an Objective-C protocol. It contains an initialization method taking a timestamp as parameter, so that during creation of a conga graph, all of the graph's nodes can be initialized with the same specific start time. To keep the interface independent of the number of input ports a node takes, there is a method to set the node's input ports with an array of conga port objects as parameter. All conga nodes are implemented to take the number of input ports they need, starting with the first array element and ignoring surplus array

elements. If a node is provided with less input ports than it needs, it will simply not carry out the processing it is intended to. To keep the node interface independent of the number of output ports a node has, there is a method that reports the concrete number of output ports of the node plus a method that is given a number as input and returns the corresponding output port of the node or nil if there is no corresponding output port. So the way conga nodes are implemented, input ports are supplied to a node all at once while output ports can only be acquired from a node one by one. Last but not least, the node interface contains a method that can be called to update all output ports of a node for the current time.

The common interface of all conga ports is defined as an Objective-C protocol as well and is kept very simple, consisting of only one method to evaluate the port. This method receives a timestamp of the current time as input and delivers the value of the port and the estimated origination time of the value as output. For nodes with several output ports, conga provides the CONGASimpleNodeOutputPort class that has to be provided with a reference to its host node and implements the port evaluation method so that it causes the host node to update all of its output port values and value origination times as soon as the first output port is evaluated for a new timestamp. As this step of indirection would represent unnecessary processing overhead in the case of conga nodes with only one output port, these nodes do not use CONGASimpleNodeOutputPort objects. Instead, they implement the conga port interface and thus serve as their own output port.

One final thing to keep in mind is the fact that conga has been developed for processing data at regular time intervals. In other words, there are some conga nodes that rely on their data input sequence being made up of input values that are more or less evenly spaced in time. For example, nodes that calculate weighted averages of successive input values might produce meaningless results otherwise. Input data provided to the starting nodes of a conga graph should be resampled, if it is not evenly spaced in time, and the timestamps of evaluation requests should be evenly spaced as well.

## 4.5 Some Examples of Basic Processing Nodes

In the previous section, conga's basic processing model has been discussed. This section demonstrates how the concepts mentioned in the previous section have been realized in conga by giving some examples of actual conga nodes. Working through all of the nodes contained in the conga framework would require more space than is appropriate, without going beyond the essential insights already provided by the given examples.

The first example, CONGAPassiveValueNode, embodies the first of the three fundamental node types, the one that just holds data. It is followed by several examples of the node type that simply processes data and passes it on, with varying functions and input and output port counts. CONGADetectZeroCrossingNode serves as an example of the third node type, the one that analyzes data for a certain feature and indicates if the feature is present. Finally, CONGANotNode shows an example of a node that treats numeric values as boolean values. Such logic nodes are needed because conga does not have a separate data type for boolean values.

### 4.5.1 CONGAPassiveValueNode

**Input ports taken:**
CONGAPassiveValueNode objects do not take any input ports.

**Output ports provided:**
CONGAPassiveValueNode objects provide only one output port.

**Processing carried out:**
As CONGAPassiveValueNode objects do not take any input ports, they perform no processing of input values. They just store a number as well as the estimated origination time of that number and present those when being queried for their current values.

**Possible use:**
CONGAPassiveValueNode objects are best suited as starting nodes of a conga graph or as nodes providing constant values needed by other nodes. Code using conga graphs can change the values stored in a CONGAPassiveValueNode object at any time, but should refrain from doing so while the corresponding graph is being evaluated. If a CONGAPassiveValueNode object is used as a source for a constant value, it can be configured to present the timestamp of the evaluation request as estimated origination time instead of the stored origination time value.

### 4.5.2 CONGAOnePoleFilterNode

**Input ports taken:**
CONGAOnePoleFilterNode objects take one input port.

**Output ports provided:**
CONGAOnePoleFilterNode objects have one output port.

**Processing carried out:**
CONGAOnePoleFilterNode objects represent infinite impulse response one-pole filters [Coo02]. In essence, their output value is calculated as the sum of the current input value multiplied with a certain factor and the last output value multiplied with another factor. The estimated origination time value of the output port is simply set to the current one of the input port.

**Possible use:**
CONGAOnePoleFilterNode objects can be configured by setting the input value and output value factors. Depending on these factors, they can be used as high-pass filter or as low-pass filters, but with the wrong factor values, they can become unstable as well. As low-pass filters they might be applied to reduce noise.

### 4.5.3 CONGAAdderNode

**Input ports taken:**
CONGAAdderNode objects take two input ports.

**Output ports provided:**
CONGAAdderNode objects provide one output port.

**Processing carried out:**
CONGAAdderNode objects add the current values supplied by their two input ports and set the value of their output port to the resulting sum. They compare the current origination time values of both input ports and use the older one as origination time value for their output port.

**Possible use:**
CONGAAdderNode objects are used for the very simple arithmetic operation of adding two numbers. But even simple arithmetic operations have to be wrapped in the guise of conga nodes so they can be incorporated into a conga graph. The simple graph from Section 4.4 that calculates $5 + \frac{x}{y}$ as one of its results, produces that result with the aid of a CONGAAdderNode object.

### 4.5.4 CONGAMaximumNode

**Input ports taken:**
CONGAMaximumNode objects take an arbitrary number of input ports, but because they compare the values of their input ports to produce output, they should be given at least two input ports in order to be useful.

**Output ports provided:**
CONGAMaximumNode objects provide two output ports. The first output port supplies the maximum of their input port values and the second one holds the index of the input port carrying said value.

**Processing carried out:**
CONGAMaximumNode objects select the maximum of all the current input port values and set the value of their first output port to this maximum value. The value of their second output port is set to the index of the input port that carries the maximum value and the estimated origination time value of this input port is passed on to both output ports. If multiple input ports carry the maximum value, the one with the most recent origination time value gets selected. Index counting starts with zero, that is, the first input port is assigned an index of zero.

**Possible use:**
CONGAMaximumNode objects can simply be used to determine the maximum of several values. But they can also be used to choose from different alternatives, if it is possible to represent the quality of each of the alternatives as a characteristic number that grows with quality.

### 4.5.5 CONGASwitchNode

**Input ports taken:**
CONGASwitchNode objects take an arbitrary number of input ports. In order to be useful they should be supplied with at least three input ports, though.

**Output ports provided:**
CONGASwitchNode objects provide two output ports, with the actual node output carried by the first output port and the second output port indicating if this is a regular output.

**Processing carried out:**
CONGASwitchNode objects use the current value of their first input port to select one of the other input ports and pass the current value and estimated origination time value of this input port on to their first output port. The value supplied by their first input port gets truncated and the result is interpreted as index of one of the remaining input ports. That is, an index value of zero selects the second input port, an index value of one selects the third input port, and so on.

If there is no corresponding input port for the index value, the first output port is set to the current value and estimated origination time value of the first input port. CONGASwitchNode objects indicate with their second output port if their index input is valid. That is, if there is an input port corresponding to the index value, the second output port value is set to one and otherwise it is set to zero. The origination time value of the second output port is simply set to the same value as the one of the first output port.

**Possible use:**

CONGASwitchNode objects supplement nodes that choose from alternatives but cannot output all data associated with each alternative. For example if a CONGAMaximumNode object is used to select an alternative based on a characteristic number, as mentioned in CONGAMaximumNode's description, it can only output the characteristic number and the selected index. But needed data of the selected alternative can still be obtained by feeding the selected index and the data of all the alternatives into a CONGASwitchNode object in the correct order. If multidimensional data has to be handled this way, several instances of CONGASwitchNode can be employed.

CONGASwitchNode produces boolean values as part of its output but it is no feature detection node in the strict sense.

### 4.5.6 CONGAHysteresisNode

**Input ports taken:**

CONGAHysteresisNode objects take an arbitrary number of input ports.

**Output ports provided:**

CONGAHysteresisNode objects provide one output port for each input port they have been supplied with.

**Processing carried out:**

CONGAHysteresisNode objects consider their input port and output port values to be the values of a vector's elements. They keep their output steady as long as the current input vector is within a certain distance of the output vector. If the distance between input vector and output vector exceeds a preset distance, the output vector is set to the values and origination times of the current input vector. The distance between the two vectors is calculated according to the Euclidean metric.

**Possible use:**

CONGAHysteresisNode objects can be used to reduce noise because they suppress changes in multidimensional data below a certain threshold. They also work on one-dimensional data, so unlike CONGAMaximumNode objects they do perform a useful function if provided with only one input port.

### 4.5.7 CONGADetectZeroCrossingNode

**Input ports taken:**
CONGADetectZeroCrossingNode objects take only one input port.

**Output ports provided:**
CONGADetectZeroCrossingNode objects provide two output ports. The first output port carries a boolean value, telling if a valid zero crossing has been detected, and the second output port holds the sign of the node's input prior to the last detected valid crossing of zero. Like other feature detection nodes, CONGADetectZeroCrossingNode uses the first output port to signal detection of the feature and the remaining output port to deliver additional information. Other feature detection nodes may have more than one output port providing additional information — or none, if they merely have to detect whether their corresponding feature is present or not.

**Processing carried out:**
CONGADetectZeroCrossingNode objects analyze the sequence of values of their input port and indicate if the values of this sequence cross zero in a certain way. If a valid zero crossing has been detected, the value of the first output port is set to one, otherwise it is set to zero. Whether or not a valid zero crossing has been detected, the estimated origination time of the first output port is always set to the current one of the input port. The second output port is only updated when a valid zero crossing has been detected and carries the sign of the last non-zero input value previous to the detected zero crossing as well as the estimated origination time of that input value.
What CONGADetectZeroCrossingNode objects consider to be a valid zero crossing can be configured in several ways. It can be set if and for how many consecutive samples the input may linger on zero. It is also possible to set a threshold value that has to be exceeded previous to crossing zero in order for the crossing to be valid, and it can be configured whether exceeding this threshold has to happen after the last crossing or the last grazing of zero, for the next zero crossing to be considered valid.

**Possible use:**
CONGADetectZeroCrossingNode objects turn out to be useful in a lot of cases. For example, if the input values are values of one component of the velocity of the tip of the baton, crossing zero indicates a reversal of direction of the baton tip's movement in this component. But if the tip of the baton is moving mainly along the x axis, noise could trigger a lot of zero crossings in the velocity's y component. The configurability of CONGADetectZeroCrossingNode objects helps to avoid detecting these inappropriate zero crossings.

### 4.5.8 CONGANotNode

**Input ports taken:**
CONGANotNode objects take one input port.

**Output ports provided:**
CONGANotNode objects provide a single output port.

**Processing carried out:**
CONGANotNode objects regard the numeric value supplied by their input port as a boolean value, negate this boolean value and output a numeric value representing the result. That is, if the current input port value is zero, the output port value is set to one. If the current input port value is not zero, the output port value is set to zero. The current estimated origination value of the input port is simply passed on to the output port in both cases.

**Possible use:**
CONGANotNode objects can be used to negate the boolean output of a feature detector node. Like CONGAAdderNode, CONGANotNode represents a simple operation that has to be implemented as a conga node in order to be usable inside a conga graph.

## 4.6 Beat Pattern Tracking

Basic processing and analysis of the trajectory of the conductor's baton or right hand can be achieved with the concepts introduced in the previous section. But conga contains more advanced parts to aid the tracking of conducting gestures. In particular, there are components to handle the first beat of a beat pattern and components to model beat patterns and to track their execution.

### 4.6.1 Handling the First Beat

The first beat of a beat pattern deserves special attention because it is the most important beat in a measure[3], and it also serves to separate subsequent beat pattern cycles. The trajectory of a conducting gesture before the first beat is very recognizable. Nearly all of the beat patterns with several beats given in Max Rudolf's book show the same trajectory leading up to the first beat: The baton moves up and left, then turns straight down, giving the trajectory a spike, and the location of the first beat is at the lowest point of the movement straight down. The trajectories of the 1-beat patterns only differ from this trajectory leading up to the first beat in that they do not feature movement to the left while moving up. Of course, every beat is a first beat when conducting a 1-beat pattern, so the described part of the trajectory in most cases represents a complete cycle of a 1-beat pattern.

A description of the conga feature detector that helps handling the first beat in a measure follows below. This node, called CONGADetectBeat1Candidates2DNode, is built to detect the very distinct part of the trajectory of the beat patterns leading up to the first beat. Not every detected candidate for the first beat does have to be a first beat, though, because the characteristic trajectory can occur in other parts of some beat patterns as well.

There are, however, no specialized beat detectors in conga for the other beats in a measure because the parts of the trajectories corresponding to other beats vary a lot more in different beat patterns. Depending on the beat pattern, different feature detectors are suited for detection of beats other than the first one.

#### CONGADetectBeat1Candidates2DNode

**Input ports taken:**

CONGADetectBeat1Candidates2DNode objects take two input ports. They interpret the values of the first input port as velocity values of the x component of the movement of the tip of the baton and the values of the second input port as velocity values of the respective y component.

[3] At least normally it is. A measure is the smallest group of notes, defining what number of notes of which length make up such a group. Most of the time, a beat pattern is chosen that covers one measure and has a compatible number of beats (often the number of beats and notes in a measure are the same). If this is the case, the first beat is the most important beat in a measure. But there are some cases when one cycle of the beat pattern does not correspond to one measure, for example a 1-beat pattern can be chosen with each (first) beat corresponding to a note.

**Output ports provided:**

CONGADetectBeat1Candidates2DNode objects provide three output ports. The first output port signals detection of a possible candidate for the first beat and the second output port carries how far the baton tip was moving down previous to the last detection of a candidate for the first beat. The third and last output port supplies the time it took the tip of the baton to move down to the last detected candidate for the first beat.

**Processing carried out:**

CONGADetectBeat1Candidates2DNode objects detect if the baton tip moved up, moved straight down afterwards and then came to a halt. Positive velocities of the y component correspond to moving up, negative velocities correspond to moving down. So CONGADetectBeat1Candidates2DNode objects wait for the y velocity to become positive at first, then negative and then zero or positive again. If at this point the previous movement down was straight, then and only then the first output port is set to a value of one (otherwise it is set to a value of zero) and the origination time of all three output ports is set to the one of the current y velocity input. In addition, the second output port value is is set to the distance the baton tip covered while moving down. The third output port value is set to the time delta between the time of the current y velocity input and the last y velocity value previous to moving down. A downward movement is considered to be straight if the distance it covered in y is at least the distance it covered in x multiplied by a certain factor, which has a default value of three and can be set to other values. To calculate the distance a downward movement covers, CONGADetectBeat1Candidates2DNode objects sum the absolute values of all velocity values provided by the input ports during the downward movement.

CONGADetectBeat1Candidates2DNode objects can be configured to indicate detection of a candidate for the first beat not immediately when the baton halts after moving down but instead when it is moving up again. This way robustness increases because the node then always places the first beat at the lowest point of the downward movement, but latency increases as well.

**Possible use:**

CONGADetectBeat1Candidates2DNode objects are rather specialized. They are intended to be used to detect candidates for the first beat and are probably of not much use for other purposes. If a 1-beat pattern is conducted, they might be sufficient to track the complete pattern, using the second and third output port values to derive volume and tempo information. Other beat patterns require more complex components to model and track them, but those components will still use CONGADetectBeat1Candidates2DNode objects to help them detect the first beat of the pattern.

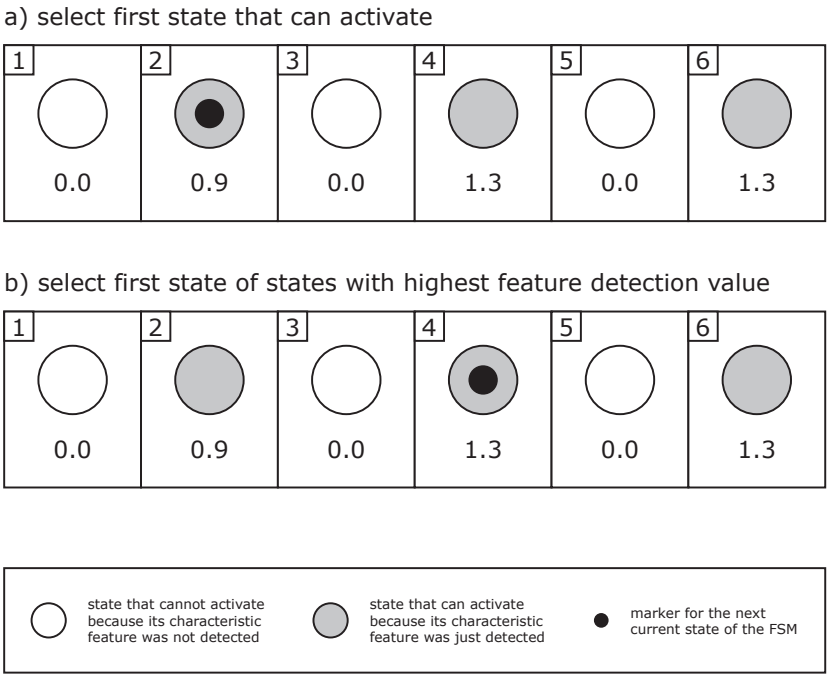


Figure 4.5: Two strategies for choosing the next active state of the conga FSM. The numbered slots represent the ordered list of possible successor states of the state that is active at the moment. The values underneath the state symbols are the values reported by their corresponding feature detectors.

### 4.6.2 Modeling and Tracking the Cycle of a Beat Pattern

For modeling the complete cycle of a beat pattern, conga provides a package of a custom finite state machine and states suited to this special form of finite state machine. The basic idea is that distinct features of the trajectory of a beat pattern occur at more or less fixed times in the cycle of this beat pattern. For example, in a 4-beat pattern executed at an even tempo, the first beat marks the start of the pattern, the second beat occurs after a quarter of the time it takes to complete the pattern, the third beat occurs at half the time, the fourth at three quarters of the time and the next first beat concludes the cycle and starts a new one. Detecting the characteristic features then allows to follow the conductor as he progresses through the beat pattern. To represent points in the cycle of a beat pattern, conga uses the FSM states. Each state object is given a value between zero and one, corresponding to its point in the gesture cycle, and a feature detector that tells, if its characteristic feature is present in the conducting input.

The complete cycle of a beat pattern is represented as a network of connected states, where each state has one or several states that are legal successor states. A conga FSM goes through this cycle by stepping from one state to the next. That is, if the last detected characteristic feature corresponded to a certain state, causing it to become the FSM's active state, it is then valid to assume progress in the beat pattern up to the

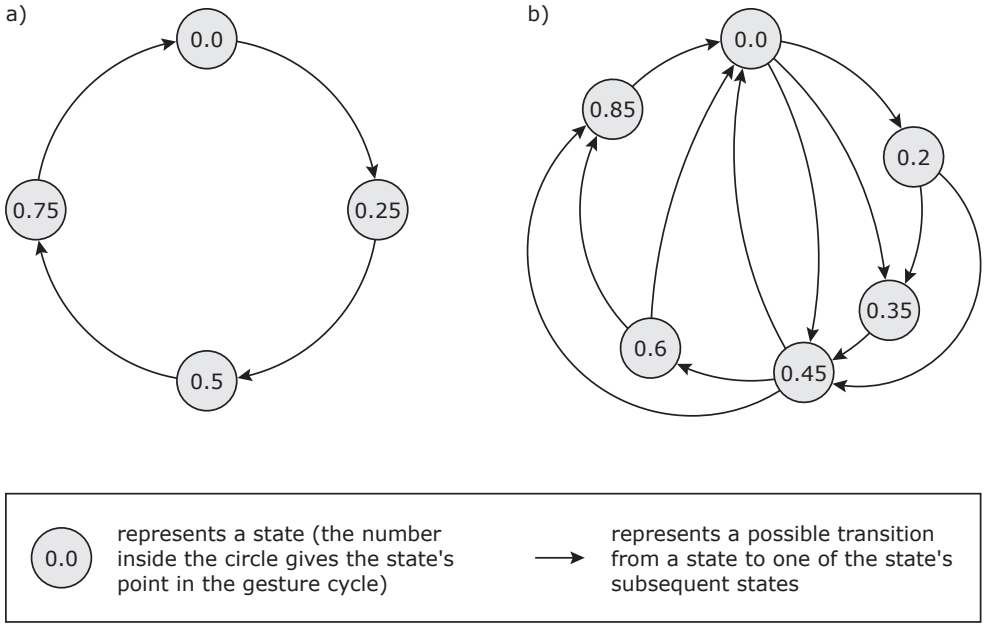


Figure 4.6: Examples of conga state networks. The left one shows an idealized network for a 4-beat pattern. The right one shows a more realistic but still hypothetical network.

point in the cycle of a successor state of said state, if that successor state detects its characteristic feature. Furthermore, it is only possible to advance in a beat pattern, it is not possible to go back. This means that activation of a state by the FSM starts the next beat pattern cycle, if the cycle position of this state is equal to or less than the one of the previously active state[4].

In conga, states and FSM work hand in hand to track a beat pattern, the states are not passive. The FSM stores which state is the active state and controls when a transition to a successor state can take place. The active state controls which state becomes the next active state. Figure 4.5 shows the two strategies that can be employed by the active state to choose the next active state from the list of its successor states. It can simply take the first one that is able to activate, or it can take the values of the corresponding feature detectors into account and go for the first one of the states with the highest feature detection value.

Examples of conga state networks representing beat patterns can be seen in Figure 4.6. The network on the left side is an idealized model for a 4-beat pattern, assuming that the feature detectors of the states always detect their characteristic feature without fail and that the features corresponding to the beats are the ones that are easiest to recognize. The network on the right side is a more realistic but still hypothetical network. If it was

[4]In theory, any FSM advance could cover more than one complete cycle of the beat pattern. In practice, the feature detectors or the state network of the affected gesture tracker need to be revised, if that happens too often.

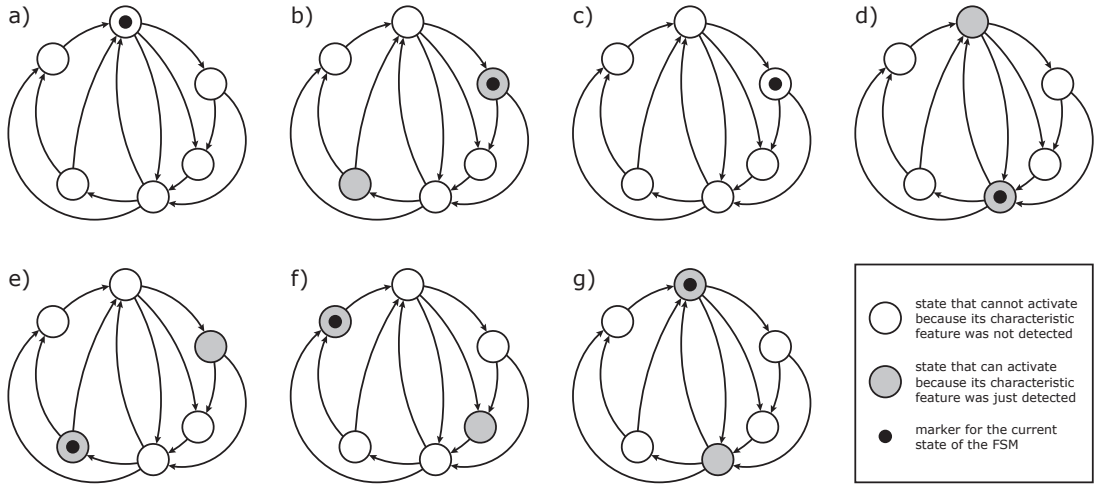


Figure 4.7: Hypothetical tracking of one of the beat pattern state networks from Figure 4.6.

a network for a 4-beat pattern, only the first beat would coincide with a state[5]. The network allows to skip states because sometimes features will not be detected properly. Figure 4.7 shows how this beat pattern might be tracked, if several states would be able to activate on detection of the same feature[6]. In Figure 4.7, the states at times 0.0 and 0.45 in the cycle correspond to a feature A, the states at times 0.2 and 0.6 correspond to a feature B and the states at times 0.35 and 0.85 correspond to a feature C:

a) The FSM starts in the state placed at time 0.0.

b) Then feature B is detected and the state at time 0.2 becomes the new active state.

c) After that, the feature detector for the state at time 0.35 misses feature C, so the active state of the FSM does not change.

d) Next, feature A is detected and the state at time 0.45 becomes the active state, skipping the state at time 0.35.

e) Afterwards, detection of feature B leads to the state at time 0.6 becoming the new active state.

f) Then feature C is detected and the state at time 0.85 becomes the new active state.

g) When feature A gets detected, one cycle in the execution of the beat pattern is completed as the state at time 0.0 becomes the active state.

[5]For example, in the 4-beat neutral-legato pattern from Figure 4.1 it is easier to detect the spikes than to detect the beats that do not fall on spikes.

[6]Very similar features *can* occur multiple times in one cycle of a beat pattern. In Figure 4.1, the 4-beat neutral-legato pattern shows a partial trajectory directly after the first beat that looks like a scaled down version of the partial trajectory immediately before the first beat.

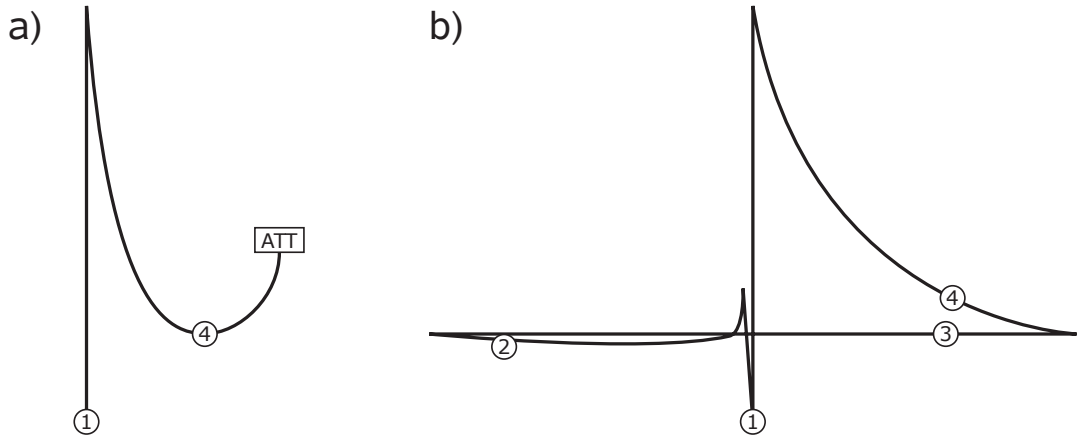


Figure 4.8: a) Trajectory for the start of a piece on the first beat, using the fourth beat in 4-beat legato as preparatory beat. b) Trajectory of the cycle of the 4-beat neutral-legato pattern. Drawings after [Rud95].

### 4.6.3 How to Enter the Cycle of a Beat Pattern

Conductors start conducting a piece of music by giving a preparatory beat to set the tempo. The baton or the right hand travels to this preparatory beat from a special position of attention. As a result, the trajectory leading up to this beat from the position of attention does not have the same shape as the trajectory leading up to this beat from the previous beat when the beat pattern is executed repeatedly. Figure 4.8 gives an example of this phenomenon for the 4-beat legato patterns, with the piece starting on the first beat and the fourth beat being used as preparatory beat. Comparing this preparatory beat trajectory with the trajectory of the normal cycle of the 4-beat neutral-legato shows that the trajectory features for a preparatory beat can differ from the trajectory features of the same beat when the pattern is executed repeatedly. Moreover, conductors do not even have to start or resume conducting a piece on the first beat of a measure.

To accommodate this, the FSM provides a way to enter the cycle of a beat pattern without using one of the states that make up this cycle. A state network for a beat pattern can include state chains that lead into the cycle of the beat pattern but cannot be reached from states that are part of the cycle. The start points of these state chains are also used by the FSM if it was already tracking a beat pattern, has been stopped and then should pick up tracking the same beat pattern as conducting resumes. For this, the start state and pick-up states are assigned a point in time of the beat pattern cycle and the FSM picks up tracking with the pick-up state whose point in time comes directly before the last known position of the FSM in the cycle. Figure 4.9 shows a hypothetical state network of a 4-beat pattern, with pick-up states for each beat.

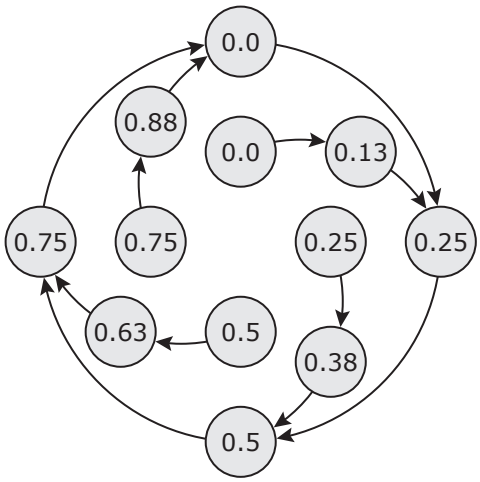


Figure 4.9: Separate start/pick-up states allow to handle starting a musical piece or picking up conducting after a stop differently from handling the execution of the cycle of the corresponding beat pattern.

Instead of being stopped, the FSM can be paused as well, with the effect that no state transitions take place as long as the FSM is paused, but the cycle of the beat pattern state network will not be left either. After the end of the pause, tracking of the beat pattern resumes with the same state that was active before the pause began.

### 4.6.4 Actual Conga Components for Finite State Machine and States

The actual conga classes implementing FSM and states have been designed to be conga nodes so that they can be placed in a conga graph much like other nodes, taking their input data from parts of the graph and allowing other parts of the graph to process their output. However, they differ from normal nodes because they are of not much use as separate components. An FSM object without a state network is useless, a state object that is not associated with an FSM does not improve on what its feature detector can do. So FSM and states have to be combined and the process of combining them cannot be mapped properly on the process of combining normal conga nodes. conga defines Objective-C protocols for the interfaces of FSM and states that cover the methods needed for them to be set up, to be combined and to interact.

The interface of conga FSM states is defined in the CONGAState protocol. It contains a method to set a state's successor states, providing those states inside an array in their correct order. Code that creates a beat pattern tracker will use this method. Then there is a method to be used by the FSM to ask the active state which, if any, of its successor states can become the next active state for the current timestamp. This method either returns nil, if no successor state can activate, or it returns the chosen successor state, as well as its activation value, the corresponding origination time and its current position

in the cycle of the beat pattern[7]. To implement this method, states can use another method of the interface that reports for the current timestamp the activation value of a state as given by its feature detector, the corresponding origination time value and the current position in the beat pattern's cycle. Also included in the interface is a method that allows the FSM to tell a state whether it is the active state or not. Finally there is a method that simply returns the current position in the beat pattern's cycle for the current timestamp. This method is used by an FSM object to query start and pick-up states for their cycle position as these states normally do not occur as successors of other states. CONGAState does not define methods to set a state's feature detector or position in the cycle. As states are meant to be conga nodes as well, these will be supplied via input ports to the state nodes.

The interface of conga finite state machines for beat pattern tracking is defined in the CONGAFiniteStateMachine protocol. It features several methods to configure an FSM and several methods to request information while tracking a beat pattern. On the configuration side, there is a method to provide the FSM with an array containing all state objects needed to track a certain beat pattern. The order of the state objects in the array does not matter, but states are also nodes and often will be the end points of a conga graph with no further nodes connected to their output ports. Because of the basic processing model of conga, all state nodes have to be evaluated each time step, and the FSM does so. There is also a method to set the first active state of the FSM, i.e., its start state. Another method supplies the FSM with an array containing all states that should serve as pick-up states after a stop. When picking up after a stop, the FSM is meant to select the pick-up state with the cycle position that is closest but after the cycle position of the active state before the stop. In addition, there is a method to set a minimum amount of time that has to pass after a state became active state before the next state can be activated, based on the origination times of the activation values of both states. All other methods of CONGAFiniteStateMachine can be used to request information concerning the tracking of a beat pattern, such as if the tracking did advance for the current time stamp, if an advance started a new cycle of the pattern, when did the activation of the active state originate, what is the cycle position of the active state, what was the difference of origination times of the activations of the current active state and the previous one and what was the corresponding distance of their cycle positions. But as conga FSMs are conga nodes as well, it is not necessary to use those methods in order to track a beat pattern. A conga FSM node has to provide all tracking information via its output ports, and stopping or pausing an FSM node should be done via input ports supplied to it.

A description of the conga nodes implementing the state and FSM protocols follows.

[7] The position of a feature in the cycle of a beat pattern might vary slightly, for example if the beat pattern is conducted at different speeds or by different conductors. By not fixing the cycle position of a state, conga allows to build conducting gesture trackers that can be adapted without having to be reinitialized. Cycle positions that shift too much can pose a problem, though: conga's FSM model assumes that a new cycle has been started, if the current cycle position of the state becoming the active state marks an earlier point in the cycle than the cycle position the last active state had, when it was activated.

### CONGASimpleState

**Input ports taken:**
CONGASimpleState objects take two input ports. As states represent points in beat pattern trajectories, a CONGASimpleState object needs a feature detector to signal detection of the characteristic feature of the point it represents. This is done via the first input port. It also needs the position of the point in the cycle of the beat pattern, which is supplied by the second input port.

**Output ports provided:**
CONGASimpleState objects provide three output ports. The first output port indicates if the state is the active state of the corresponding FSM, the second carries the value the state activated with and the third supplies the cycle position held by the state when it became the active state.

**Processing carried out:**
For processing, CONGASimpleState objects need a corresponding FSM, or else their output will never change. When a CONGASimpleState object is evaluated as a conga node, it just queries both input ports and caches the current value provided by its feature detector, the associated origination time and the cycle position value. Each time the corresponding FSM is itself evaluated, it will first evaluate all of its state nodes, ignoring their output but causing them to update their cached values, and then decide whether or not to switch the active state. In the process, the current active state requests the cached values of its successor states to determine which one of them, if any, it will present to the FSM as possible next active state. If the FSM switches states, it deactivates the old active state and activates the new one. Only this causes a change in the output of both states. The freshly activated state sets the first output port value to one, the second output port value to the current value reported by its feature detector and the third output port value to its current cycle position. The other state only updates the first output port with the value zero. Both states use a time value provided by the FSM as estimated origination time for all output ports that they updated. In the case of CONGASimpleFiniteStateMachine, this is simply the origination time value reported by the feature detector of the state that becomes the new active state.
To avoid inconsistencies in the caching mechanism of conga's basic processing, CONGASimpleState objects change their output port values after being activated or deactivated not until they get evaluated with a more recent timestamp. Because they are conga nodes, they might have been evaluated for a certain timestamp by other nodes before their corresponding FSM is evaluated for the same timestamp. If they changed their output port values immediately and were afterwards evaluated again for the current timestamp, they would have supplied different output for the same timestamp, thereby violating the basic processing model.

**Possible use:**
CONGASimpleState objects are meant to be used to model and track beat patterns in combination with a conga FSM. They employ the simply strategy to choose the next possible active state from their successor states by selecting the first one that can activate, as shown in part a) of Figure 4.5, and should be used whenever this state transition behavior is appropriate.

### CONGASelectSuccessorByValueState

**Input ports taken:**
CONGASelectSuccessorByValueState objects take two input ports.

**Output ports provided:**
CONGASelectSuccessorByValueState objects provide three output ports.

**Processing carried out:**
Processing done by CONGASelectSuccessorByValueState and CONGASimpleState objects is very similar, only differing in the strategy used to determine the next possible active state of the corresponding FSM.

**Possible use:**
CONGASelectSuccessorByValueState objects choose the next possible active FSM state from their successor states by comparing the feature detection values of the successor states, as shown in part b) of Figure 4.5. They can be used if the feature detector for a state's characteristic feature is able to tell how strong this feature is and not only if it is present or not. It is possible to mix CONGASelectSuccessor-ByValueState and CONGASimpleState objects in the state network of an FSM.

### CONGASimpleFiniteStateMachine

**Input ports taken:**
A CONGASimpleFiniteStateMachine object takes three input ports. The first input port indicates if the FSM should be stopped, the second one indicates if it should be paused and the third one indicates if it should be restarted.

**Output ports provided:**
A CONGASimpleFiniteStateMachine object provides seven output ports. The first output port signals if the FSM advanced in the cycle of the beat pattern it models. The second one tells if the last advance started a new cycle. The third one holds the feature detection value that caused the active state to be activated. The fourth output port carries the current position of the FSM in the cycle. The fifth one supplies the distance the FSM moved forward in the cycle with the last advance. The sixth one provides the difference of the activation times of the current active state and the previous one. The final output port tells how many cycles have already been tracked by the FSM.

**Processing carried out:**

Evaluation of a CONGASimpleFiniteStateMachine causes it to query the input ports and update all its state objects. If the third input port signals a restart, the FSM resets internal values, activates its start state and updates all output ports, using the current timestamp as estimated origination time. Neither does this count as advance in the cycle, nor does it start a new cycle. The activation value of the start state is defined to be zero. The FSM's current position in the cycle is set to the one provided by the start state and the position delta is set to zero. Activation time delta is set to one, mainly to avoid a division by zero if other components use it to calculate tempo. Finally, the number of cycles already tracked by the FSM is set to minus one, because zero would correspond to the first cycle and the preparatory beat to start a piece lies outside the first cycle.

If no restart takes place but the input ports indicate that the FSM should be stopped or paused, only the first output port is updated. Its value is set to zero and its origination time is set to the one of the first input port when stopping or of the second input port when pausing the FSM. If the FSM stops, it prepares to pick up tracking again by activating the correct pick-up state.

In case the FSM does not stop or pause, it requests from the active state a successor state that could activate. If there is no such successor state, the FSM sets the first output port value to zero and the associated origination time to the current timestamp. It does the same if the the difference between the origination time value provided by the successor state's feature detector and the activation time value of the current active state is less than the FSM's minimum time between advances. Otherwise the FSM uses the activation value, the associated origination time value and the current cycle position of said successor state to calculate and store the relevant information for its output. Then it activates the successor state and deactivates the previously active state. Finally, the FSM signals an advance in the cycle by setting the first output port value to one and all the other output ports to their appropriate values, passing on to all output ports the origination time value reported by the new active state.

Start and pick-up states usually correspond to positions of attention, consequently their activation is not considered to be an FSM advance. Normally, their successor states correspond to a preparatory beat or something like that, so these are still positioned outside the cycle of a beat pattern. Therefore, CONGASimpleFiniteStateMachine treats their activation as silent advance. A silent advance is handled like a normal advance, but it cannot start a new cycle and and the first output port does not signal it as advance.

**Possible use:**

CONGASimpleFiniteStateMachine is a very specialized conga node, intended to model and track beat patterns according to conga's model of conducting gestures, aided by conga state objects. Appendix A describes a component that transforms cycle count and cycle position into an absolute time value inside a musical piece, useful for some forms of output of computer-based conducting systems.

# 5 Using the Framework

This chapter gives two examples how conga can be used. It presents as first example the simple graph from Section 4.4, which, despite probably being of not much real use, is well suited to demonstrate graph creation and evaluation with a short piece of source code. In contrast to this very simple graph, the second example is taken from a working tracker for the 4-beat neutral-legato pattern. The code to create and configure this beat pattern tracker is a lot more complex, even though it clearly shows repeating patterns. Any useful tracker of a 4-beat pattern will be about as complex or even more complex than the second example.

## 5.1 Source Code for Figures 4.3 and 4.4

The simple graph mentioned in Section 4.4 calculates $5 + \frac{x}{y}$ and $3\frac{x}{y}$. Figure 4.3 shows how this graph might be created and Figure 4.4 shows how it might be evaluated. The following Objective-C source code realizes those figures:

```
// graph creation according to Figure 4.3
float now = 0.0f;


// Figure 4.3 part a)
CONGAPassiveValueNode *xNode;
xNode = [[CONGAPassiveValueNode alloc] initWithTime:now];

CONGAPassiveValueNode *yNode;
yNode = [[CONGAPassiveValueNode alloc] initWithTime:now];


// Figure 4.3 part b)
CONGADividerNode *dividerNode;
dividerNode = [[CONGADividerNode alloc] initWithTime:now];
[dividerNode inputPorts:[NSArray arrayWithObjects:xNode, yNode, nil]];


// Figure 4.3 part c)
CONGAPassiveValueNode *constant5Node;
constant5Node = [[CONGAPassiveValueNode alloc] initWithTime:now];
[constant5Node useCurrentTimeAsTimeOfValue:YES];
[constant5Node value:5.0f];
```

```
CONGAPassiveValueNode *constant3Node;
constant3Node = [[CONGAPassiveValueNode alloc] initWithTime:now];
[constant3Node useCurrentTimeAsTimeOfValue:YES];
[constant3Node value:3.0f];


// Figure 4.3 part d)
CONGAAdderNode *adderNode;
adderNode = [[CONGAAdderNode alloc] initWithTime:now];
[adderNode inputPorts:
 [NSArray arrayWithObjects:constant5Node, dividerNode, nil]];

CONGAMultiplierNode *multiplierNode;
multiplierNode = [[CONGAMultiplierNode alloc] initWithTime:now];
[multiplierNode inputPorts:
 [NSArray arrayWithObjects:constant3Node, dividerNode, nil]];



// graph evaluation according to Figure 4.4
float resultA, timeA;
float resultB, timeB;

[xNode value:4.0f];
[yNode value:2.0f];
now = 1.0f;

[adderNode getValue:&resultA andTimeOfValue:&timeA forCurrentTime:now];
[multiplierNode getValue:&resultB andTimeOfValue:&timeB forCurrentTime:now];

NSLog(@"5 + x/y = %f, 3*x/y = %f", resultA, resultB);

// prepare next graph evaluation
[xNode value:3.0f];
[xNode timeOfValue:now];
[yNode value:8.0f];
[yNode timeOfValue:now];
now = 2.0f;
```

In the code above, all nodes have only a single output port. Normally, output port objects of a node are acquired with a call of the node's `outputPort:` method, but conga is implemented so that node objects with only one output port serve as their own output port object, so the corresponding port objects and lines like `[someNode outputPort:0];` to acquire them have been left out of the sample code.

## 5.2 Simple Tracker for 4-Beat Neutral-Legato Pattern

It is possible to track the 4-beat neutral-legato pattern by using five characteristic features of its trajectory, as shown in Figure 4.1: The partial trajectory leading to the first beat, the baton changing direction in y after going up from the first beat, the baton changing direction in x after the second and third beat plus the spike after the fourth beat[1]. Detecting these features is easier than trying to detect all of the beats directly. Figure 5.1 illustrates the conga graph that analyzes if those features are present in an input sequence of position values of the tip of the baton. The states present in Figure 5.1 get connected to form a state network, and this network is inserted into an FSM object to track the beat pattern, as depicted in Figure 5.2.
All components needed by this simple tracker can be created and configured as follows:

```
startTime = 0.0f;

// input nodes
xNode = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
yNode = [[CONGAPassiveValueNode alloc] initWithTime:startTime];

// nodes for smoothing input and calculating velocities
smoothXNode = [[CONGAHanningSmoothingNode alloc] initWithTime:startTime];
[smoothXNode setBufferSize:31];
[smoothXNode inputPorts:[NSArray arrayWithObject:xNode]];
smoothYNode = [[CONGAHanningSmoothingNode alloc] initWithTime:startTime];
[smoothYNode setBufferSize:31];
[smoothYNode inputPorts:[NSArray arrayWithObject:yNode]];

smoothDeltaXNode = [[CONGADeltaNode alloc] initWithTime:startTime];
[smoothDeltaXNode inputPorts:[NSArray arrayWithObject:smoothXNode]];
smoothDeltaYNode = [[CONGADeltaNode alloc] initWithTime:startTime];
[smoothDeltaYNode inputPorts:[NSArray arrayWithObject:smoothYNode]];


// partial graph up to feature detector for first beat
hysteresisNode = [[CONGAHysteresisNode alloc] initWithTime:startTime];
[hysteresisNode inputPorts:[NSArray arrayWithObjects:xNode, yNode, nil]];
[hysteresisNode distanceThreshold:(2.0f/127.0f)];
xPort = [hysteresisNode outputPort:0];
yPort = [hysteresisNode outputPort:1];

filteredXNode = [[CONGAOnePoleFilterNode alloc] initWithTime:startTime];
[filteredXNode inputPorts:[NSArray arrayWithObject:xPort]];
[filteredXNode currentInputValueWeight:0.1f];
[filteredXNode previousOutputValueWeight:0.89f];
```

[1] This tracker can also track the 4-beat expressive-legato pattern from Figure 4.1 because it does not search for spikes in the trajectory after the second and third beat. But the two beat patterns do not position those features at the very same points in their cycles, so timing will be slightly off if the tracker is configured for one of both patterns and then used to track the other one.

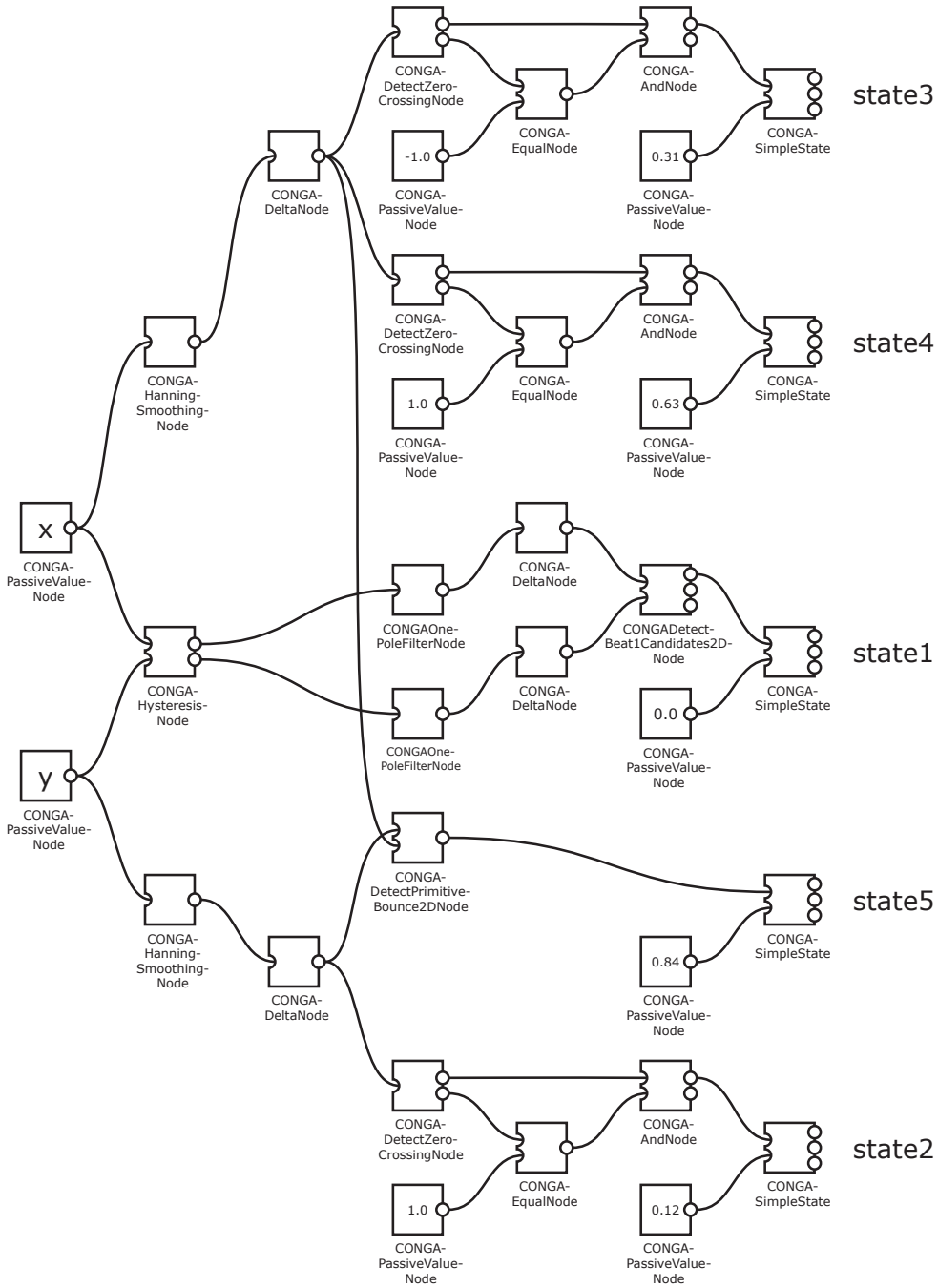


Figure 5.1: conga graph for detecting characteristic features in the 4-beat neutral-legato pattern's trajectory. The states needed to track the pattern are included, but not the FSM because it is not linked to this graph as conga node.

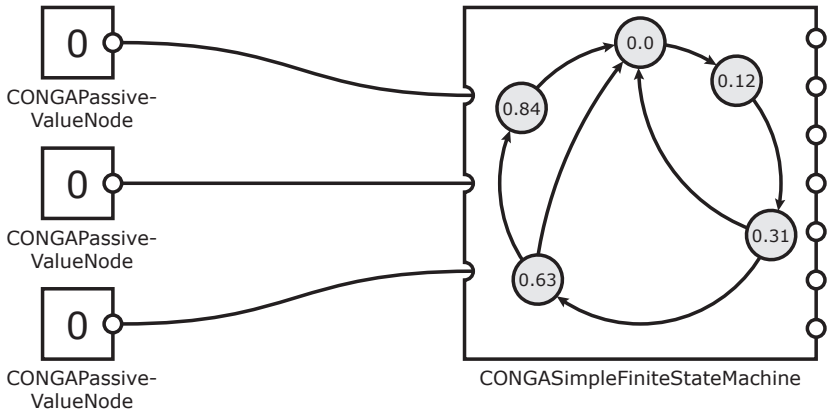


Figure 5.2: conga FSM with state network for tracking the 4-beat neutral-legato pattern.

```
filteredYNode = [[CONGAOnePoleFilterNode alloc] initWithTime:startTime];
[filteredYNode inputPorts:[NSArray arrayWithObject:yPort]];
[filteredYNode currentInputValueWeight:0.1f];
[filteredYNode previousOutputValueWeight:0.89f];

filteredDeltaXNode = [[CONGADeltaNode alloc] initWithTime:startTime];
[filteredDeltaXNode inputPorts:[NSArray arrayWithObject:filteredXNode]];
filteredDeltaYNode = [[CONGADeltaNode alloc] initWithTime:startTime];
[filteredDeltaYNode inputPorts:[NSArray arrayWithObject:filteredYNode]];

detectBeat1Node = [[CONGADetectBeat1Candidates2DNode alloc]
                   initWithTime:startTime];
[detectBeat1Node inputPorts:
 [NSArray arrayWithObjects:filteredDeltaXNode, filteredDeltaYNode, nil]];
[detectBeat1Node waitForUpwardMotionToAcknowledgeBeat1Candidate:YES];
detectBeat1Port = [detectBeat1Node outputPort:0];


// partial graph up to feature detector for turnaround in y after first beat
detectZeroCrossingDYNode = [[CONGADetectZeroCrossingNode alloc]
                            initWithTime:startTime];
[detectZeroCrossingDYNode threshold:0.0015f];
[detectZeroCrossingDYNode grazingZeroCancelsExceedingThreshold:YES];
[detectZeroCrossingDYNode limitStayAtZero:YES];
[detectZeroCrossingDYNode maxCountInputStaysAtZero:5];
[detectZeroCrossingDYNode inputPorts:[NSArray arrayWithObject:smoothDeltaYNode]];
detectZeroCrossingDYPort = [detectZeroCrossingDYNode outputPort:0];
signZeroCrossingDYPort = [detectZeroCrossingDYNode outputPort:1];

signValueNodeState2 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[signValueNodeState2 value:1.0f];
[signValueNodeState2 useCurrentTimeAsTimeOfValue:YES];
```

```
checkSignNodeState2 = [[CONGAEqualNode alloc] initWithTime:startTime];
[checkSignNodeState2 inputPorts:
 [NSArray arrayWithObjects:signZeroCrossingDYPort, signValueNodeState2, nil]];

detectorNodeState2 = [[CONGAAndNode alloc] initWithTime:startTime];
[detectorNodeState2 inputPorts:
 [NSArray arrayWithObjects:detectZeroCrossingDYPort, checkSignNodeState2, nil]];
detectorPortState2 = [detectorNodeState2 outputPort:0];


// partial graph up to feature detector for turnaround in x after second beat
detectZeroCrossingDXNode1 = [[CONGADetectZeroCrossingNode alloc]
                             initWithTime:startTime];
[detectZeroCrossingDXNode1 threshold:0.0015f];
[detectZeroCrossingDXNode1 grazingZeroCancelsExceedingThreshold:NO];
[detectZeroCrossingDXNode1 limitStayAtZero:YES];
[detectZeroCrossingDXNode1 maxCountInputStaysAtZero:10];
[detectZeroCrossingDXNode1 inputPorts:[NSArray arrayWithObject:smoothDeltaXNode]];
detectZeroCrossingDXPort1 = [detectZeroCrossingDXNode1 outputPort:0];
signZeroCrossingDXPort1 = [detectZeroCrossingDXNode1 outputPort:1];

signValueNodeState3 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[signValueNodeState3 value:-1.0f];
[signValueNodeState3 useCurrentTimeAsTimeOfValue:YES];

checkSignNodeState3 = [[CONGAEqualNode alloc] initWithTime:startTime];
[checkSignNodeState3 inputPorts:
 [NSArray arrayWithObjects:signZeroCrossingDXPort1, signValueNodeState3, nil]];

detectorNodeState3 = [[CONGAAndNode alloc] initWithTime:startTime];
[detectorNodeState3 inputPorts:
 [NSArray arrayWithObjects:detectZeroCrossingDXPort1, checkSignNodeState3, nil]];
detectorPortState3 = [detectorNodeState3 outputPort:0];


// partial graph up to feature detector for turnaround in x after third beat
detectZeroCrossingDXNode2 = [[CONGADetectZeroCrossingNode alloc]
                             initWithTime:startTime];
[detectZeroCrossingDXNode2 threshold:0.002f];
[detectZeroCrossingDXNode2 grazingZeroCancelsExceedingThreshold:NO];
[detectZeroCrossingDXNode2 limitStayAtZero:YES];
[detectZeroCrossingDXNode2 maxCountInputStaysAtZero:25];
[detectZeroCrossingDXNode2 inputPorts:[NSArray arrayWithObject:smoothDeltaXNode]];
detectZeroCrossingDXPort2 = [detectZeroCrossingDXNode2 outputPort:0];
signZeroCrossingDXPort2 = [detectZeroCrossingDXNode2 outputPort:1];

signValueNodeState4 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[signValueNodeState4 value:1.0f];
[signValueNodeState4 useCurrentTimeAsTimeOfValue:YES];
```

```
checkSignNodeState4 = [[CONGAEqualNode alloc] initWithTime:startTime];
[checkSignNodeState4 inputPorts:
 [NSArray arrayWithObjects:signZeroCrossingDXPort2, signValueNodeState4, nil]];

detectorNodeState4 = [[CONGAAndNode alloc] initWithTime:startTime];
[detectorNodeState4 inputPorts:
 [NSArray arrayWithObjects:detectZeroCrossingDXPort2, checkSignNodeState4, nil]];
detectorPortState4 = [detectorNodeState4 outputPort:0];


// feature detector for spike after fourth beat
detectBounceNode = [[CONGADetectPrimitiveBounce2DNode alloc]
                    initWithTime:startTime];
[detectBounceNode minRatioAccumulationFirstInputToAccumulationSecondInput:4.0f];
[detectBounceNode accumulationBufferSize:23];
[detectBounceNode threshold:0.001f];
[detectBounceNode grazingZeroCancelsExceedingThreshold:YES];
[detectBounceNode limitStayAtZero:NO];
[detectBounceNode inputPorts:
 [NSArray arrayWithObjects:smoothDeltaYNode, smoothDeltaXNode, nil]];


// the five states and their position nodes
cyclePositionNodeState1 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[cyclePositionNodeState1 value:0.0f];
[cyclePositionNodeState1 useCurrentTimeAsTimeOfValue:YES];
state1 = [[CONGASimpleState alloc] initWithTime:startTime];
[state1 inputPorts:
 [NSArray arrayWithObjects:detectBeat1Port, cyclePositionNodeState1, nil]];

cyclePositionNodeState2 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[cyclePositionNodeState2 value:0.12f];
[cyclePositionNodeState2 useCurrentTimeAsTimeOfValue:YES];
state2 = [[CONGASimpleState alloc] initWithTime:startTime];
[state2 inputPorts:
 [NSArray arrayWithObjects:detectorPortState2, cyclePositionNodeState2, nil]];

cyclePositionNodeState3 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[cyclePositionNodeState3 value:0.31f];
[cyclePositionNodeState3 useCurrentTimeAsTimeOfValue:YES];
state3 = [[CONGASimpleState alloc] initWithTime:startTime];
[state3 inputPorts:
 [NSArray arrayWithObjects:detectorPortState3, cyclePositionNodeState3, nil]];

cyclePositionNodeState4 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[cyclePositionNodeState4 value:0.63f];
[cyclePositionNodeState4 useCurrentTimeAsTimeOfValue:YES];
state4 = [[CONGASimpleState alloc] initWithTime:startTime];
[state4 inputPorts:
 [NSArray arrayWithObjects:detectorPortState4, cyclePositionNodeState4, nil]];
```

```
cyclePositionNodeState5 = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[cyclePositionNodeState5 value:0.84f];
[cyclePositionNodeState5 useCurrentTimeAsTimeOfValue:YES];
state5 = [[CONGASimpleState alloc] initWithTime:startTime];
[state5 inputPorts:
 [NSArray arrayWithObjects:detectBounceNode, cyclePositionNodeState5, nil]];


// state network
[state1 subsequentStates:[NSArray arrayWithObjects:  state2, nil]];
[state2 subsequentStates:[NSArray arrayWithObjects:  state3, nil]];
[state3 subsequentStates:[NSArray arrayWithObjects:  state4, state1, nil]];
[state4 subsequentStates:[NSArray arrayWithObjects:  state5, state1, nil]];
[state5 subsequentStates:[NSArray arrayWithObjects:  state1, nil]];


// dummy FSM input nodes - this example has no conga graphs that decide
// whether or not the FSM should be stopped, paused or restarted
stopFSMNode = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[stopFSMNode value:0.0f];
[stopFSMNode useCurrentTimeAsTimeOfValue:YES];

pauseFSMNode = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[pauseFSMNode value:0.0f];
[pauseFSMNode useCurrentTimeAsTimeOfValue:YES];

restartFSMNode = [[CONGAPassiveValueNode alloc] initWithTime:startTime];
[restartFSMNode value:0.0f];
[restartFSMNode useCurrentTimeAsTimeOfValue:YES];

// FSM for tracking the beat pattern
neutralLegato4BeatPatternFSM = [[CONGASimpleFiniteStateMachine alloc]
                                initWithTime:startTime];
[neutralLegato4BeatPatternFSM inputPorts:
 [NSArray arrayWithObjects:stopFSMNode, pauseFSMNode, restartFSMNode, nil]];
[neutralLegato4BeatPatternFSM allStates:
 [NSArray arrayWithObjects:state1, state2, state3, state4, state5, nil]];
[neutralLegato4BeatPatternFSM pickupAfterStopStates:
 [NSArray arrayWithObject:state4]];
[neutralLegato4BeatPatternFSM startState:state4];
[neutralLegato4BeatPatternFSM minimumTimeBetweenAdvances:0.1];
fsmDidAdvanceForCurrentTimePort = [neutralLegato4BeatPatternFSM outputPort:0];
fsmLastAdvanceStartedNewCyclePort = [neutralLegato4BeatPatternFSM outputPort:1];
fsmActivationValueLastAdvancePort = [neutralLegato4BeatPatternFSM outputPort:2];
fsmPositionInCyclePort = [neutralLegato4BeatPatternFSM outputPort:3];
fsmPositionDeltaLastAdvancePort = [neutralLegato4BeatPatternFSM outputPort:4];
fsmTimeDeltaLastAdvancePort = [neutralLegato4BeatPatternFSM outputPort:5];
fsmCycleCountPort = [neutralLegato4BeatPatternFSM outputPort:6];
```

Everything needed for processing conducting gesture input has to be configured in the code above, making it rather extensive. In contrast to the code creating and setting up all conga components used in the tracker, the code to actually track the beat pattern by repeatedly evaluating those components is comparatively simple:

```
now = currentSystemTime - systemTimeAtInitialization;

// feed input data into graph
[xNode value:currentBuchlaXInput];
[xNode timeOfValue:(now - buchlaLatency)];
[yNode value:currentBuchlaYInput];
[yNode timeOfValue:(now - buchlaLatency)];


// get tracking output from FSM
[fsmDidAdvanceForCurrentTimePort getValue:&didAdvance
                                 andTimeOfValue:&didAdvanceTime
                                 forCurrentTime:now];
[fsmLastAdvanceStartedNewCyclePort getValue:&newCycle
                                   andTimeOfValue:&newCycleTime
                                   forCurrentTime:now];
[fsmActivationValueLastAdvancePort getValue:&activationValue
                                   andTimeOfValue:&activationValueTime
                                   forCurrentTime:now];
[fsmPositionInCyclePort getValue:&positionInCycle
                        andTimeOfValue:&positionInCycleTime
                        forCurrentTime:now];
[fsmPositionDeltaLastAdvancePort getValue:&positionDelta
                                 andTimeOfValue:&positionDeltaTime
                                 forCurrentTime:now];
[fsmTimeDeltaLastAdvancePort getValue:&timeDelta
                             andTimeOfValue:&timeDeltaTime
                             forCurrentTime:now];
[fsmCycleCountPort getValue:&cycleCount
                   andTimeOfValue:&cycleCountTime
                   forCurrentTime:now];
```

The FSM's tracking output allows to derive the conducting tempo during the last advance from `positionDelta` and `timeDelta`. To calculate the conducted volume, the second output port of `detectBeat1Node` could be used: each time an advance of `neutralLegato4BeatPatternFSM` starts a new cycle, the value of this output port gives the height of the last cycle's conducting gesture, indicating the volume. Also, when a new cycle starts, the output port values of `filteredXNode` and `filteredYNode` approximate the position of the first beat in the beating plane, which might be useful to determine what instrument group the conductor focuses on.

To keep the tracker simple, there are no facilities to stop, pause or restart the FSM. Consequently, there are also no states outside the normal cycle of the beat pattern.

All code in this example is taken from a program working on Buchla Lightning II baton input data, with position values of the baton rescaled to values between zero and one. One thread of this program takes in and stores baton data, while another thread is run repeatedly every 9 ms, supplying the conga components with the current baton data and evaluating them in order to track the beat pattern[2]. All time values that are used in the code are measured in seconds.

It would be possible to change the resampling rate or to switch to another input device with different noise characteristics, without having to change the structure of this tracker. This allows to encapsulate the whole tracker into a single conga node for later reuse. But the configuration parameters of the following nodes would probably have to be changed in order to achieve acceptable tracking results for each input device:

- `smoothXNode`
- `smoothYNode`
- `hysteresisNode`
- `filteredXNode`
- `filteredYNode`
- `detectZeroCrossingDYNode`
- `detectZeroCrossingDXNode1`
- `detectZeroCrossingDXNode2`
- `detectBounceNode`

Thus the interface of the conga node representing the whole tracker would have to include methods for configuring all of these nodes.

[2]Resampling at 9 ms and using a one-pole filter as configured in the code are ideas taken from Guy Garnett's Adaptive Conductor Follower (presented in Subsection 2.2.7). The ideas work for this example, because the Adaptive Conductor Follower also used a Buchla baton system as input hardware.

# 6 Conclusions and Future Work

In summary, conga was intended to be an input hardware independent framework that helps to create and reuse components that can process, analyze, recognize and track conducting gestures of the conductor's baton or right hand.

conga uses simple numbers as input data and can pass on associated latency of an input device. The framework mainly consists of processing nodes that can be connected to form directed acyclic graphs, which serve to process and analyze conducting input. It includes a special finite state machine and corresponding states to model and track beat patterns. Combined with conga graphs that detect characteristic features of the trajectory of a conducting gesture, they allow recognition and tracking of conducting gestures. Components built with conga can be reused by encapsulating them in a conga graph node or in an Objective-C class, and they can be adapted to changing requirements by reconfiguring the nodes of their conga graphs or, if that is not sufficient, by restructuring these graphs. conga components can be mixed with C++ and Objective-C source code to form a single Mac OS X application.

## 6.1 Conclusions

The conga framework meets a lot of the goals for its development, but not all of them.

On the positive side, conga as framework is independent of the input hardware that is used, and it does enable and aid the construction of components to process, analyze, recognize and track conducting gestures of the conductor's baton or right hand, as long as input and gestures conform to conga's model of conducting gestures as two-dimensional trajectories of a point. The framework might even be useful for at least processing input that does not conform to its conducting gesture model, if the processing can be modeled as directed acyclic graph of processing units and said input consists of simple numbers.

conga also supports reuse of components implemented with it: they can be encapsulated as Objective-C classes for direct reuse in other projects, or they can be encapsulated as conga nodes to make them reusable inside a conga graph. If these container components provide access to the configuration methods of the encapsulated components, they can be adapted to changed requirements, if these requirements can be met without changing the internal structure of the container components.

One important strength of conga is that it does not force a programer to use it for everything. If some processing cannot be expressed easily as a directed acyclic graph and consequently is hard to model with conga, it can be implemented without using conga and then be combined with conga graphs that perform the processing conga is suited

for. Creating glue code to combine conga and non-conga components is comparatively simple: Either a wrapper node has to be built in order to be able to embed the non-conga component in a conga graph, or the glue code has to split the conga graphs where necessary, passing data directly between conga graphs and non-conga components and evaluating all of them in the correct order.

Processing in conga is definitely fast enough to handle real-time conducting input. One of the testing programs created by the author during development of conga reads recorded Buchla Lightning II baton input data from a file, resamples the recorded data at time intervals of 9 ms and feeds the resampled data into the simple beat pattern tracker presented in Section 5.2. When executed on an Apple iBook notebook with 320 MB of RAM and a G3 processor running at 500 MHz, all processing done by the tracker in this simulated tracking of conducting input needs less than two percent of the time span covered by the recorded data. Therefore, more recent hardware like a PowerMacintosh with 512 MB of RAM and dual G5 processors running at 2 GHz, as used during development of conga, should be able to run several more complex trackers in parallel plus a demanding algorithm for sound synthesis, like a phase vocoder.

On the negative side, although conga itself is independent of the input hardware that is used, modules to recognize and track conducting gestures that are implemented with conga components are not. If such a module has been configured for one input hardware and is then switched to another one, it will probably have to be reconfigured if the new input hardware differs significantly from the old one in important characteristics like noise. In some cases it might be possible to preprocess the input of the new hardware to mimic the characteristics of the old one, for example by trading reduced noise for increased latency, adjusting the origination times of the values passed on to the original module to reflect the added latency.

Building beat pattern trackers from scratch takes a lot of time. One not only has to figure out which conga components to use and how to connect them, but also how to configure all those components, with each conga node influencing its subsequent nodes in a conga graph. Even a simple tracker like the one from Section 5.2 is complex enough to make fine-tuning it a lengthy process of trial and error. So conga aids the construction of components to track conducting gestures, but conga does not make it easy.

conga includes optimizations that improve processing speed. Because these optimizations add complexity to the source code of conga components and also influence how code is executed, they impede debugging of the framework's components and they impede extending the framework.

Despite the amount of work that has gone into conga, it does not feel finished yet. Section 4.6 mentions conga components that model conducting-specific concepts, but there should be more such components for other conducting-specific concepts. For example, it is possible to use conga to construct gesture trackers that distinguish between staccato and legato gestures, but there was not enough time to develop and include components in conga that correspond to staccato or legato features of the trajectory of a conducting gesture.

## 6.2 Future Work

In the application domain of conducting, conga can be improved in several ways. The node to detect candidates for the first beat in a beat pattern can be split up into several nodes that are more specific to the beat patterns: the trajectory leading to the first beat in patterns with multiple beats differs from the trajectory used in 1-beat patterns and there is a difference between the staccato and legato version of this trajectory. conga might also benefit from being extended with components incorporating neural networks and hidden markov models. Using neural networks or HMMs to recognize complete beat patterns will often result in the need to train them to individual conductors. But if they are applied to distinct features of a beat pattern's trajectory, it could turn out that there exist features that do not need to be retrained. Again, the partial trajectory leading to the first beat is a possible candidate for such an improvement.

The current version of conga focuses on conducting gestures performed with the right hand, which are well suited for time-beating and can indicate musical expression as well. An opportunity for future research would be to expand conga in the direction of less formal input, like expressive gestures of the left hand or even patterns of physiological signals, as covered in the research of Paul Kolesnik ([Kol04]) and Teresa Marrin ([Nak00]), respectively. Less formal input might give the conductor more expressive control, but the important signals probably vary from person to person. Tracking them will thus probably require components that can be trained to an individual conductor, like the already mentioned neural networks and HMMs.

Another promising area to extend conga is the handling of latency. A system using the Buchla Lightning II as input hardware can have an overall latency in the range of 100 ms [LWB05]. For conducting, this is quite high, and conga does nothing to reduce latency, it simply passes on the accumulated latency. Components that predict features instead of detecting them could be added to conga to alleviate this problem. If these components present the predicted time when their associated features will occur instead of passing on the origination time of their current input sample, they can be used in gesture trackers without the existing FSM and state classes having to be modified. Furthermore, if such a component calculates the prediction value as a probability value, this value will grow as the time approaches that the corresponding feature actually occurs, making it possible to use a threshold to balance out latency and reliability. The paper *A New Control Paradigm: Software-Based Gesture Analysis for Music* by Ben Nevile, Peter Driessen and W. A. Schloss applies similar ideas to the Radio Drum [NDS03].

A more fundamental area of future work on conducting gestures concerns the question of where exactly conductors place the beats. The placement of some beats in the beat patterns according to Max Rudolf does not correspond to a position where the trajectory of the tip of the baton has reached its lowest point, yet several of the conducting systems presented in Section 2.2 assume that *all* beats sit at locally lowest points of the conducting gesture's trajectory. To find out where trained conductors place the beats, a study could survey conductors either conducting to a metronome or conducting while saying out loud which beat they are currently conducting. It would be beneficial, if different schools of conducting were represented in this study with several conductors

each, to see if the different schools agree on the placement of beats, or if at least all conductors of each school place beats the same way. In addition, the participants should conduct a variety of commonly used beat patterns, each in several tempi, and it should be analyzed how this influences beat placement. For optimal results, the study should record the conductors with input devices that have high temporal and spatial resolution and precision.

There are also promising extensions to conga that do not focus on conducting. For example, LabVIEW, EyesWeb and Max/MSP provide graphical programming environments and allow to do all programming graphically. Because conga graphs and non-conga components can be mixed freely in source code, it is not possible to create a graphical programming environment for conga that simply reads in source code that uses conga components, allows to change those graphically and then writes the results back to source code without ever breaking the non-conga components. But it is possible and would be worthwhile to create a graphical programming environment that handles only conga components and includes the ability to transform the graphic representations of conga graphs into source code. As the sample code in Section 5.2 shows, code to create and configure a conga graph is rather schematic anyway, so building a source code generator for conga graphs should not be too hard. Such a graphical programming environment could speed up the development of conga graphs, at least up to the point where non-conga components have to be added.

Computers still keep getting faster and their performance will increase for at least a few years, a trend that might render the speed optimizations in conga obsolete. The ever growing processing power could also make it feasible to implement conga in a platform-independent language like Java. But that would only be of interest, if demanding forms of output for computer-based conducting systems could be implemented in the same language, in a platform-independent way, and still be executed without glitches. Otherwise one would be forced to split the handling of input and output of such a system into separate applications, something conga was intended to help avoid in the first place.

Some computer-based conducting systems track the conductor's baton in three dimensions, but conga is built to aid tracking of the baton in two dimensions, because most of the information contained in a baton gesture can be represented in two dimensions of space and one dimension of time. It would be easy to extend conga to support three-dimensional tracking, so it could be applied in domains where fully three-dimensional gesture input is useful. Mobile computing might become one of those domains as cell phone makers already plan to equip their phones with sensors to acquire gesture input from three-dimensional movements of the phone[1].

Another application domain outside conducting where conga could be useful might be pen computing. conga employs a special variant of a finite state machine because while a certain beat pattern is being conducted, the same gesture occurs again and again without interruption. By turning the FSM into a finite automaton, conga could be

[1] Samsung Electronics introduced their mobile phone SCH-S310 in January 2005, featuring sensors and algorithms to recognize three-dimensional movements. The press release can be found at http://www.samsung.com/AboutSAMSUNG/ELECTRONICSGLOBAL/InvestorRelations/NewsPublicDisclosure/PressRelease/PressRelease.asp?seq=20050112_0000096403

used to implement modules that recognize gesture input with the gestures occurring as separate entities. In pen computing this could be used to interpret each sequence of pen positions between the pen being put down and being lifted up again as a gesture. Several automata standing for different input gestures could be evaluated in parallel and the one accepting the complete input sequence would determine which gesture is recognized. Afterwards, conga graphs associated with the recognized gesture could process the same input sequence, to derive parameters from it that provide further information to an action triggered by the recognition of this gesture.

# A Interpolating Beat Times

conga allows to track a beat pattern using points in the trajectory of the beat pattern that do not correspond to beats, and the output of a conga FSM does not say where in the cycle of a beat pattern the beats are placed. It does, however, tell where the active state of the FSM is placed in this cycle and CONGASimpleFiniteStateMachine also outputs how many cycles have already been completed. With this information, it is easy to calculate which beat of the beat pattern was the last conducted beat and how many beats have been conducted in total. But some forms of output of a computer-based conducting system not only need to know how many beats have already been conducted. They need to know the exact time the current position of the gesture tracker corresponds to in the piece being conducted. For example if a score format is used that just stores notes and the times they are played or if an audio and video recording of a real orchestra is played back, each advance of the beat pattern tracker should yield the corresponding time in the score or in the movie. conga provides a node that works on an array containing the absolute times of all beats in the musical piece and that can interpolate between these times based on the output of CONGASimpleFiniteStateMachine.

## A.1 CONGABeatTimeInterpolatorNode

**Input ports taken:**
CONGABeatTimeInterpolatorNode objects take two input ports. The first input port value is interpreted as the number of complete beat pattern cycles already conducted and the second input port value is interpreted as a position inside the current beat pattern cycle.

**Output ports provided:**
CONGABeatTimeInterpolatorNode objects provide one output port that supplies the interpolated absolute time value.

**Processing carried out:**
Before they can be used, CONGABeatTimeInterpolatorNode objects have to be provided with an array containing the absolute time values of all beats in ascending order. Per default, they assume that there are four beats in each cycle, but this can and must be changed, if a beat pattern uses a different number of beats. When a CONGABeatTimeInterpolatorNode object is evaluated, it queries its first input port for the number of full cycles that have already been conducted and its second

input port for the position inside the current cycle, cropping the value if it is less than zero or greater than one. These values allow to calculate the index of the last conducted beat in the array of all beats, as well as the point between this beat and the next, assuming all beats to be evenly spaced in time. As the beats do not have to be evenly spaced in time according to their absolute time values, CONGABeatTimeInterpolatorNode then uses linear interpolation to transform the ideal point between those two beats into an absolute time value between their absolute time values. Then it sets its output port value to this interpolated absolute time value and passes the oldest origination time value of its two input ports on to its output port. If the input values reference a point outside the beat array, the output port value is set to the time of either the first or the last beat, whichever is closest.

**Possible use:**

CONGABeatTimeInterpolatorNode objects can be used the way described at the beginning of this appendix. They can also be abused as simple linear interpolators by setting the number of beats in a cycle to one and filling the array of beat times with whatever values one wishes to interpolate. This enables to interpolate two successive array entries by providing the index of the first entry to the first input port of CONGABeatTimeInterpolatorNode and the point of interpolation to the second input port.

# Bibliography


[BC92] Graziano Bertini and Paolo Carosi: *Light baton: A System For Conducting Computer Music Performance.* In *Proceedings ICMC92*, pages 73–76. ICMA, 1992.

[BG95] Bennett Brecht and Guy E. Garnett: *Conductor Follower.* In *ICMC Proceedings 1995*, pages 185–186. ICMA, 1995.

[BM97] Richard Boulanger and Max Mathews: *The 1997 Radio Baton and improvisation modes.* In *Proceedings ICMC97*, pages 395–398. ICMA, 1997.

[BMS89] Bob Boie, Max Mathews and Andy Schloss: *The Radio Drum as a synthesizer controller.* In *Proceedings ICMC89*, pages 42–45. ICMA, 1989.

[Bor97] Jan O. Borchers: *WorldBeat: Designing a Baton-Based Interface for an Interactive Music Exhibit.* In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 131–138. ACM, 1997.

[Bor01] Jan Borchers: *A Pattern Approach to Interaction Design.* Wiley Series in Software Design Patterns. John Wiley & Sons Ltd, May 2001.

[BRF+80] W. Buxton, W. Reeves, G. Fedorkow, K. C. Smith and R. Baecker: *A Microcomputer-based Conducting System.* Computer Music Journal, 1/1980, pages 8–21.

[CCM+04] Antonio Camurri, Paolo Coletta, Alberto Massari, Barbara Mazzarino, Massimiliano Peri, Matteo Ricchetti, Andrea Ricci and Gualtiero Volpe: *Toward real-time multimodal processing: EyesWeb 4.0.* In *Proc. AISB 2004 Convention: Motion, Emotion and Cognition*, March 2004.

[CMV03] Antonio Camurri, Barbara Mazzarino and Gualtiero Volpe: *Analysis of expressive gestures in human movement: the EyesWeb expressive gesture processing library.* In *Proc. XIV Colloquium on Musical Informatics*, May 2003.

[Coo02] Perry R. Cook: *Human Computer Interface Technology - Digital Signal Processing for HCI.* http://soundlab.cs.princeton.edu/learning/tutorials/DSP/DSP.html, October 2002.

[GJE+01] Guy E. Garnett, Mangesh Jonnalagadda, Ivan Elezovic, Timothy Johnson and Kevin Small: *Technological Advances for Conducting a Virtual Ensemble.* In *ICMC Proceedings 2001*, pages 167–169. ICMA, 2001.

[GMRS99] Guy E. Garnett, Fernando Malvar-Ruiz and Fred Stoltzfus: *Virtual Conducting Practice Environment.* In *ICMC Proceedings 1999*, pages 371–374. ICMA, 1999.

[Ilm] Tommi Ilmonen: *DIVA - Digital Interactive Virtual Acoustics.* http://www.tml.hut.fi/Research/DIVA/.

[Ilm98] Tommi Ilmonen: *Tracking conductor of an orchestra using artificial neural networks.* In *STeP'98, Human and Artificial Information Processing.* Finnish Conference on Artificial Intelligence, 1998.

[Ilm99] Tommi Ilmonen: *Tracking conductor of an orchestra using artificial neural networks.* Master's thesis, Helsinki University of Technology, April 1999.

[Ins] Native Instruments: *LabVIEW FAQs.* http://ni.com/labview/faq.htm.

[IT99] Tommi Ilmonen and Tapio Takala: *Conductor Following With Artificial Neural Networks.* In *ICMC Proceedings 1999*, pages 367–370. ICMA, 1999.

[Kar05] Thorsten Karrer: *PhaVoRiT - A Phase Vocoder for Real-Time Interactive Time-Stretching.* Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, August 2005.

[KG89] David Keane and Peter Gross: *The MIDI Baton.* In *Proceedings ICMC89*, pages 151–154. ICMA, 1989.

[Kol04] Paul Kolesnik: *Conducting Gesture Recognition, Analysis and Performance System.* Master's thesis, McGill University, Montreal, June 2004.

[KW91] David Keane and Kevin Wood: *The MIDI Baton III.* In *Proceedings ICMC91*, pages 541–544. ICMA, 1991.

[LGW92] Michael Lee, Guy Garnett and David Wessel: *An Adaptive Conductor Follower.* In *Proceedings ICMC92*, pages 454–455. ICMA, 1992.

[LNB04] Eric Lee, Teresa Marrin Nakra and Jan Borchers: *You're The Conductor: A Realistic Interactive Conducting System for Children.* In *NIME 2004 International Conference on New Interfaces for Musical Expression*, pages 68–73, 2004.

[LWB05] Eric Lee, Marius Wolf and Jan Borchers: *Improving Orchestral Conducting Systems in Public Spaces: Examining the Temporal Characteristics and Conceptual Models of Conducting Gestures.* In *Proceedings of the CHI 2005 Conference on Human Factors in Computing Systems*, pages 731–740. ACM, April 2005.

[MAJ03] Declan Murphy, Tue Haste Andersen and Kristoffer Jensen: *Conducting Audio Files via Computer Vision.* In *Proceedings of the 2003 International Gesture Workshop*, 2003.

[Mar96] Teresa Anne Marrin: *Toward an Understanding of Musical Gesture: Mapping Expressive Intention with the Digital Baton.* Master's thesis, Massachusetts Institute of Technology, June 1996.

[Mat00a] Max Mathews: *CONDUCTOR PROGRAM.* http://www.csounds.com/mathews, December 2000.

[Mat00b] Max Mathews: *RADIO-BATON INSTRUCTION MANUAL.* http://www.csounds.com/mathews, December 2000.

[MHO91] Hideyuki Morita, Shuji Hashimoto and Sadamu Ohteru: *Computer Music System that Follows a Human Conductor.* IEEE Computer, 7/1991, pages 44–53.

[MM70] M. V. Mathews and F. R. Moore: *GROOVE - A Program to Compose, Store, and Edit Functions of Time.* Communications of the ACM, 12/1970, pages 715–721.

[MOH89] H. Morita, S. Ohteru and S. Hashimoto: *Computer Music System which Follows a Human Conductor.* In *Proceedings ICMC89*, pages 207–210. ICMA, 1989.

[MP97] Teresa Marrin and Joseph Paradiso: *The Digital Baton: a Versatile Performance Instrument.* In *Proceedings ICMC97*, pages 313–316. ICMA, 1997.

[MP98] Teresa Marrin and Rosalind Picard: *The Conductor's Jacket: A Device for Recording Expressive Musical Gestures.* In *Proceedings ICMC98*, pages 215–219. ICMA, 1998.

[Mur03] Declan Murphy: *Tracking a Conductor's Baton.* In *Proceedings of the 12th Danish Conference on Pattern Recognition and Image Analysis 2003*, 2003.

[MWH+90] Hideyuki Morita, Hiroshi Watanabe, Tsutomu Harada, Shuji Hashimoto and Sadamu Ohteru: *Knowledge Information Processing in Conducting Computer Music Performer.* In *ICMC Glasgow 1990 Proceedings*, pages 332–334. ICMA, 1990.

[Nak00] Teresa Marrin Nakra: *Inside the Conductor's Jacket: Analysis, Interpretation and Musical Synthesis of Expressive Gesture.* PhD thesis, Massachusetts Institute of Technology, February 2000.

[NDS03] Ben Nevile, Peter Driessen and W. A. Schloss: *A New Control Paradigm: Software-Based Gesture Analysis for Music.* In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Band 1, pages 360–363. IEEE, August 2003.

[Rud95] Max Rudolf: *The Grammar of Conducting: A Comprehensive Guide to Baton Technique and Interpretation*. Schirmer Books, 3rd edition, June 1995.

[Sam02] Wolfgang Samminger: *Personal Orchestra: Interaktive Steuerung synchroner Audio- und Videoströme*. Master's thesis, Johannes Kepler Universität Linz, September 2002.

[SFMB01] Johan Sundberg, Anders Friber, Max V. Mathews and Gerald Bennett: *Experiences of combining the Radio Baton with the Director Musices performance grammar*. In *MOSART - Workshop on Current Research Directions in Computer Music*, 2001.

[SMG00] Jakob Segen, Aditi Majumder and Joshua Gluckman: *Virtual Dance and Music Conducted by a Human Conductor*. EUROGRAPHICS '2000, 3/2000.

[Sol] Realtime Music Solutions: *Realtime Music Solutions*. http://www.rms.biz/index.htm.

[Spo01] Bruno Spoerri: *Dirigieren mit Trommelschlegeln - Max Mathews, Vater der Computermusik*. Neue Zürcher Zeitung, Ressort Medien und Informatik, 97/2001, page 85.

[TF96] Forrest Tobey and Ichiro Fujinaga: *Extraction of Conducting Gestures in 3D Space*. In *ICMC Proceedings 1996*, pages 305–307. ICMA, 1996.

[Tob95] Forrest Tobey: *The Ensemble Member and the Conducted Computer*. In *Proceedings ICMC95*, pages 529–530. ICMA, 1995.

[UM98] Satoshi Usa and Yasunori Mochida: *A Multi-Modal Conducting Simulator*. In *Proceedings ICMC98*, pages 25–32. ICMA, 1998.

[Vol03] Gualtiero Volpe: *Computational models of expressive gesture in multimedia systems*. PhD thesis, University of Genova, April 2003.

All internet URLs given here or anywhere else in this thesis were up to date in April 2005. Due to the nature of the internet, there is no guarantee that these URLs stay valid afterwards, and even if they do, the contents of the corresponding websites may still change.

## Application of Tools in the Creation of this Diploma Thesis

Several tools were used to create this diploma thesis. Typesetting was done with LaTeX. All illustrations produced for this thesis have been drawn with Adobe Illustrator. Images and illustrations taken from referenced works have been processed using Adobe Photoshop. This processing was done to improve reproduction of said illustrations and images, including steps like slight changes of coloring or contrast of certain elements or whole pictures, noise reduction and marginal cropping.
No photomontage was done to change the contents of pictures. No elements have been removed or added apart from combining images into a single PDF file for easier placement in LaTeX.

Name: Ingo Grüll Matrikel-Nr.: .....

## Erklärung

Ich erkläre, daß ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ..... .....
(Unterschrift)