



City Research Online

City, University of London Institutional Repository

Citation: Honing, H. (1991). Music and the representation of structure: From issues to microworlds. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/29143/>

Link to published version:

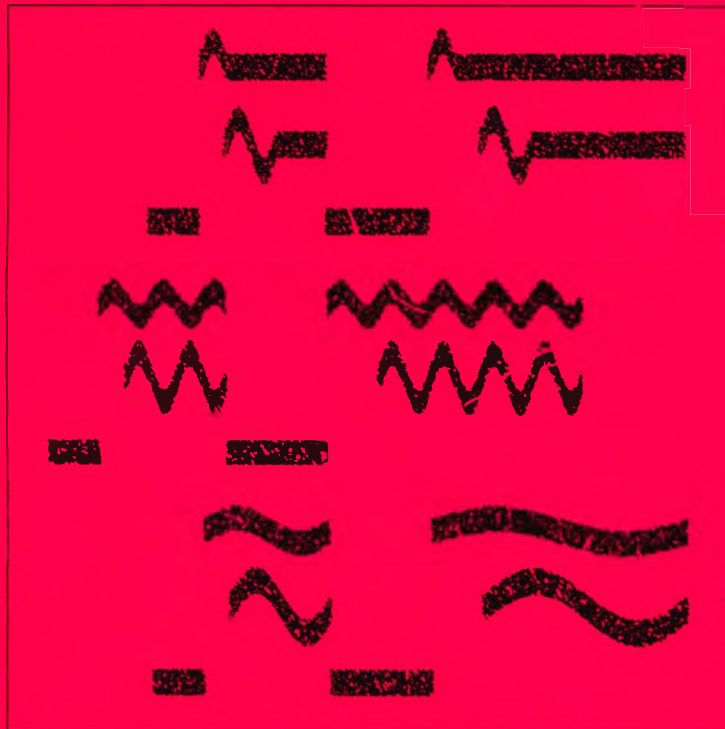
Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

MUSIC AND THE REPRESENTATION OF STRUCTURE

From issues to microworlds

Henkjan Honing

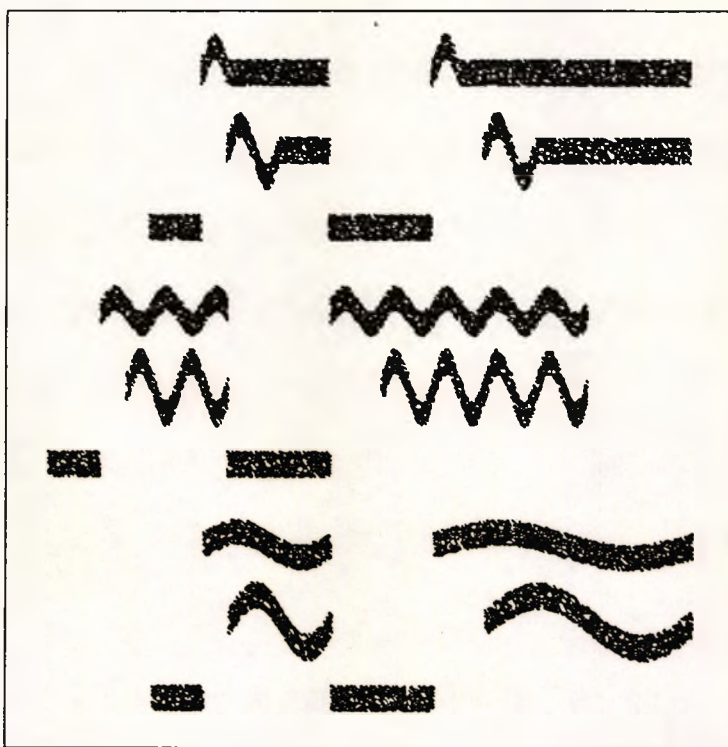


Submitted for the degree of Doctor of Philosophy in Music
City University, London
July 1991

MUSIC AND THE REPRESENTATION OF STRUCTURE

From issues to microworlds

Henkjan Honing



Submitted for the degree of Doctor of Philosophy in Music
City University, London
July 1991

**MUSIC AND
THE REPRESENTATION OF STRUCTURE**

From issues to microworlds

Henkjan Honing

I certify that all the material in this thesis which is not my own work has been identified and that no material is included for which a degree has been previously been conferred upon me.

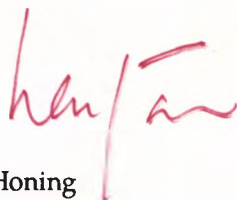
Henkjan

STATEMENT OF CO-AUTHORSHIP

Four of the six articles included in this thesis are co-authored by Peter Desain. They are the result of collaborative research over long periods of time in which we had numerous discussions, exchanged ideas, problems and solutions, tried them out together in programs and wrote the resulting articles by exchanging edited versions frequently before a final version emerged. Therefore the ideas, expressed in these four collaborative articles, are difficult to assign to either one of the authors.

We hereby declare that both "Tempo Curves considered harmful" and "Towards a calculus for expressive timing in music" are 70% the work of Henkjan Honing and 30% the work of Peter Desain. "LOCO: A composition microworld in Logo" and "Time functions function best as functions of multiple times" is 50% the work of Henkjan Honing and 50% the work of Peter Desain.

Signed Utrecht, September 10, 1991



Henkjan Honing

Peter Desain



COPYING

I grant powers of discretion to the City University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

PREFACE

The body of this thesis consists of six articles written in the last four years. They encompass research in the fields of the psychology of music, artificial intelligence and computer music. Over these years, a lot of people gave their advice, help and support. They all should be thanked here, but I will refer to the acknowledgements at the end of the respective articles for their proper credit.

Special thanks to Peter Desain, who has been an inspiring friend and colleague in the last seven years of our collaborative research. And special thanks to Eric Clarke. Firstly, as an outstanding supervisor during the writing of this thesis, and secondly, because of the support and insights he provided in the field of the psychology of music, so new to me when I started my work as a research fellow at the City University in 1988. Thanks for all the help!

Henkjan Honing

Utrecht, July 1991

ABSTRACT

This thesis is concerned with the representation of music. It investigates the issues that must be tackled when constructing a formal representation (or representational system) for expressing musical knowledge. The central problems are pointed out by examining existing proposals for the representation of music and studying models from the fields of the psychology of music and computer music research for their use of explicit or implied representations. These discussions adopt both psychological and technical perspectives and provide the essential groundwork for the formalisation of concrete solutions of a representational system for music. Partial solutions are expressed as small programs, or microworlds, that facilitate further exploration and understanding. This methodology resulted in two concrete proposals that describe separate aspects of such a representational system. The first microworld describes a generalized representation of the continuous aspects of music with respect to time. The second microworld embodies a calculus for expressive timing defined in terms of structure. Finally, two larger systems are presented that functioned as the experimental gardens of the microworlds described previously.

CONTENTS

Statement of co-authorship	i
Copying	ii
Preface	iii
Abstract	iv
Contents	v
Structure of the thesis	vi

I Introduction

II Issues

Honing, H. (1991). Issues in the Representation of Time and Structure in Music. In Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press. (forthcoming).

Desain, P. & H. Honing. (1991). Tempo curves considered harmful. In "Music and time", edited by J. D. Kramer. Contemporary Music Review. London: Harwood Press. (forthcoming)

III Microworlds

Desain, P. & H. Honing (in press). Time functions function best as functions of multiple times. To appear in Computer Music Journal. Cambridge, Mass.: MIT Press.

Desain, P. & H. Honing (1991). Towards a calculus for expressive timing in music. Research report. Utrecht: Centre for Knowledge Technology. Submitted to *Psychology of Music*.

IV Two examples of larger systems

Honing, H. (1990). POCO: An Environment for Analysing, Modifying, and Generating Expression in Music. In Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association.

Desain, P. & H. Honing (1988). LOCO: A Composition Microworld in Logo. Computer Music Journal 12(3). Cambridge, Mass.: MIT Press. 30-42.

V Conclusion

STRUCTURE OF THE THESIS

This thesis is divided into three pairs of two articles preceded by an introduction and followed by a conclusion.

I Introduction

The introduction gives, besides a description of the subject and the aim of this thesis, a detailed discussion on the methodology of building microworlds that was applied in this research. It stresses the importance of the use of a computational approach in music research, reaching from musicology to the psychology of music.

II Issues

The first pair is concerned with bringing out the issues in the representation of music.

"Issues in the representation of time and structure in music" gives a global overview of the field and the central problems that have to be solved, serving as a proper introduction to a thesis dedicated to this subject. Most of the issues are presented as controversies, using extremes to clarify the underlying problems. Associating time intervals and their constraints with the components of musical structure turns out to be essential. These constraints on time intervals model an important characteristic of musical knowledge and should be part of the representation, i.e. part of the syntax.

"Tempo curves considered harmful" is a critique of a widespread representation of time. It evaluates well-known models stemming from the literature of musicology, computer music research and the psychology of music. In this work timing or tempo measurements are mostly presented in the form of continuous curves. The article warns against the notion of 'tempo curves' as giving the false impression that a continuous concept of continuous temporal flow has an independent existence -has a musical or psychological reality- and that time can be perceived independent of the events carrying it. It is shown that when one bases a transformation or manipulation of timing on the implied characteristics of such a notion, the results do not make any musical sense.

The insights that were developed during this research yielded a second group of concrete proposals.

III Microworlds

The second pair of articles propose two concrete solutions, presented as microworlds.

"Time functions function best as functions of multiple times" is dedicated to a proposal for a representation of time. It introduces control functions that have multiple times as arguments. This generalized concept of a time function can support absolute and relative kinds of time behaviour. The possibilities of composition and transformation of time functions themselves are retained.

A microworld version of the time functions is included as an appendix.

"Towards a calculus for expressive timing in music" describes a calculus that enables expressive timing to be transformed on the basis of structural aspects of the music. The behaviour of musical material under expressive transformations is determined uniquely by its structural description and the type of expression. Although the calculus separates different kinds of behaviour, it entails no musical knowledge of the transformations themselves. It does not model music cognition, but it will hopefully prove to be a solid basis for formalised theories of music cognition.

A microworld version of the calculus is included as an appendix.

IV Two examples of larger systems

The third pair of articles describes two larger systems that formed the basis for the four preceding articles.

"POCO: an environment for analysing, modifying, and generating expression in music" describes a system that formed an important basis in comparing different existing models of expressive timing, and brought out the numerous problems that had to be solved in realising a more general representational system for music. The system integrates existing models of expression which made it possible to compare and combine these models using the same performance and score data. POCO is developed as a workbench to be used in a research context. Several tools were developed for specific "micro-surgery" on expression (including an early version of the calculus presented above). A lot of attention was given to the openness, integration, and extendability of the system.

"LOCO: a composition microworld in Logo" describes the experimental garden of the microworld approach. It was developed in the years from 1984 to 1988, and was implemented on a whole range of small machines for use by both novices, music students,

and composers. The ideal of making a composition microworld, consisting of a small set of primitives from which all kinds of higher-level compositional ideas could be constructed, is nicely illustrated in this discrete "choice-principles" microworld. The score system and the time-structuring principles that were designed for it, also proved to be valuable in later work (see Microworlds).

A more detailed description of the score system and the final descisions on a consistent set of primitives is described in the LOCO manual. It is added as an appendix.

V Conclusion

The thesis is concluded with a discussion of how the "explicit, formal, and modular strategy" can serve in working towards a general representational system for music.

I

INTRODUCTION

I

INTRODUCTION

Long-term aim.....	2
Overview of approach	2
Microworlds.....	3
Building microworlds as methodology.....	4
Moving knowledge from control structure to data structure.....	5
Is a microworld more or less than a theory?	6
Conclusion.....	7
References.....	7

INTRODUCTION

This thesis is concerned with the representation of music. It investigates the issues that must be tackled when constructing a formal representation (or representational system) for expressing musical knowledge. The central problems are pointed out by examining existing proposals for the representation of music and studying models from the fields of the psychology of music and computer music research for their use of explicit or implied representations. These discussions take both psychological and technical perspectives and provide the essential groundwork for the formalisation of concrete solutions for a representational system for music.

This thesis is about applying computational modeling and artificial intelligence techniques to music. These methods and techniques are used to evaluate and understand related research in the psychology of music and computer music research and its (partially) formalised theories. Partial solutions are expressed as small programs, or microworlds, that facilitate further exploration and understanding. They play a key role in this research. Each microworld models an isolated aspect of a representational system of music, making a certain set of issues even more explicit and broadening our understanding of it.

It is important to point out here that computational modeling is considered a powerful methodology in music cognition research, although it is clear that there are important aspects, normally associated with music, that are ignored in such an approach (e.g. the problem of embodiment). Nonetheless I will ignore the mind/machine debate. Central to this thesis is the definition of a representational *system*, not the construction of a computational *model* of music cognition and the possible internal representations used. Since the construction of a complete and general representation of music is still far ahead of us, if not fundamentally impossible, gaining understanding of what can and what cannot be represented using certain types of formal representations, is far more important. The methodology of constructing microworlds and micro-version programs has turned out to be a successful strategy in building these formalised components of a representational system, components that are well-understood and generalized in such a way that maintenance and extension is guaranteed. In the end, these formalised parts of a representational system will turn out to be a solid basis for theories of musicology, music perception and cognition.

LONG-TERM AIM

The long-term objective of this research is to develop a formal and computational representational system of music that, on the one hand, can serve as a basis for higher-level formalisms or theories, such as formalisms from musicological research intended to express specific musical knowledge (e.g. style or performance practice), theories that are based on grammars, rule-systems or stochastic formalisms, or theories of music cognition describing diverse and often isolated domains of, for instance, harmony and metre. Since all these models are based on a specific set of primitives, they will benefit from a level of abstraction that incorporates musical knowledge about these primitives (and to which new musical knowledge can simply be added). On the other hand, this anticipated representational system could make the link between lower-level models of a more perceptual and psycho physical nature, like models of rhythm and pitch perception. They could provide the "bottom-up" information to this knowledge representation layer. The possibility of realising such a representational system for music will be investigated, and the following questions will be asked. What is an effective methodology and a good strategy in realising such a system? Is it possible at all? Can we learn from experiences in other fields, e.g. knowledge representation in general? This introduction and, in more detail, this whole thesis, tries to answer these questions.

OVERVIEW OF APPROACH

The material presented in this thesis is based on experience gathered during research in computer music. What started out as an approach in designing composition systems (Desain & Honing, 1986), influenced by the work of the Logo community, developed over the years into a methodology that accompanied us in different areas of music and AI research. In this work the computer took a central position, performing different roles at the various stages of the research process. First of all, we concentrated on the construction of 'microworlds', a small and closed set of procedures and data structures. In these microworlds it is easy to experiment with ideas, vague as they are, to gain more insight into the problem to be understood and modeled. Secondly, we constructed theories in computational form. These new microworlds made the theory explicit and allowed for tests on completeness and internal consistency. Thirdly, we found much profit in (re)constructing 'micro-versions' of larger programs (or models), particularly when they were made to share the same data abstraction. In trimming these computational theories down to a "bare minimum", they allowed for better and easier comparison (Desain, 1990; Desain & Honing, 1991), that brings a real understanding of the theory with, more than once, the emergence of more abstract or general notions as a result.

A lot can be said about the advantages, disadvantages, and implications of building microworlds and micro-version programs, as a methodology. The next paragraphs look at some of the explicit and implied characteristics of this methodology.

MICROWORLDS

"What characterizes the period of the early seventies is the concept of a microworld - a domain which can be analysed in isolation." (Dreyfus, 1981)

Many of the microworld ideas stem from the Logo project (Papert, 1980/1984) and other people working at MIT in the seventies (e.g. Abelson, Minsky, Winograd, Sussman). The notion of a 'microworld' has been described by Marvin Minsky and Seymour Papert as:

"Each model - or 'micro-world' as we shall call it - is very schematic; it talks about a fairyland in which things are so simplified that almost every statement about them would be literally false if asserted about in the real world. [...] Nevertheless, we feel that they [the micro-worlds] are so important that we are assigning a large portion of our effort toward developing a collection of these micro-worlds and finding how to use the suggestive and predictive powers of the models without being overcome by their incompatibility with literal truth." (internal MIT memo Minsky & Papert, 1970; quoted in Dreyfus, 1981)

Papert and his colleagues developed several microworlds for use in an educational context, inspired by the cognitive development theory of Jean Piaget (Papert, 1981/1984). These microworlds were designed to facilitate learning, and were based on a new programming language called Logo (based on Lisp) embodying the educational philosophy of "learning without being taught."

The most prominent example of one of these microworlds is the 'turtle-world' which models a world of turtle-geometry (Abelson & diSessa, 1980). Children learned about this world by giving commands to a turtle robot, or a turtle image on a computer screen, and building procedures from them. They gained knowledge and understanding of (turtle) geometry by just exploring the possibilities of this object. These ideas had a major influence on the development of educational research and formed the basis of a widespread curriculum in computer science in primary and secondary schools.

Another, often referred to, example of the microworld notion is Winograd's block-world for natural language understanding (Winograd, 1972). Here, by contrast, the domain is not central; the microworld just serves as a toy problem to test the possibilities of a certain approach to natural language processing. This kind of microworld approach, and the

optimism that these microworlds could simply be combined and extended into a general knowledge representation, prompted a heavy critique (e.g. Dreyfus, 1981) that gave the notion of microworlds a bad press (causing Winograd to take the side of his critics, see Winograd & Flores, 1986). This critique, though, should be placed in the perspective of using microworlds to model human knowledge, instead of seeing them as part of a methodology that brings out the isolated problem under study and make it explicit (in the case of Winograd's microworld, the representation and processing of natural language). There are still strong arguments for the design and use of microworlds.

Besides these ideas normally associated with microworlds (i.e. to model a toy problem, or to facilitate "learning without being taught"), there are much broader implications that make it a valid and important notion in computational modeling and artificial intelligence research. Several aspects are involved, which will be described below.

Building microworlds as methodology

"The best way of finding out the difficulties of doing something is to try to do it." (David Marr, 1985; p. 108)

This near-cliché has, as every cliché, the reality of the obvious. But the quote illustrates well an important characteristic of AI research that stands for trying out ideas in the form of programs. Vague formalisms, parts of theories, and "poorly understood and sloppily formulated ideas" (as Marvin Minsky calls them) come up against a tough discipline in programming, the language of AI. Minsky promoted "exploratory programming" to avoid having to start with a complete and detailed specification: "an excessive preoccupation with formalism is impeding our development" (Minsky, 1987). This exploratory programming (using microworlds) was one of the key concepts in the beginning of AI in the early seventies, a newly emerging methodology, and an alternative to empirical research.

In my own work I have frequently found that actually programming a certain idea can provide new insights. It brings out other aspects of a possible solution because the program forces you to answer questions you didn't think of, or it suggests a way of programming it in another way (e.g. choosing a different data abstraction or control mechanism). A microworld, because of its relatively small dimensions, invites you to do things "completely differently" because not all the work (as in a larger system) is dependent on the abstractions chosen. Experimenting with the resulting ad hoc formalisation or program may bring out further insights, providing a real understanding, that, in turn,

possibly provides for a new formalisation, and a new theory. In making problems concrete, deciding what is essential and what isn't, and moving knowledge and understanding from being implicit (e.g. in the control structure) to being explicit (e.g. as data structures), problems become objects, objects of thought, that facilitate thinking about them - just as the turtle gave children "an object to think with" (Papert, 1980/84), helping them to understand more about geometry.

Moving knowledge from control structure to data structure

People grasp simple problems better than complex ones. Therefore computer languages have been developed which allow for convenient ways of arranging information (i.e. algorithms and data) in simple chunks that facilitate the readability and understanding of it by human beings. The complexity of a problem can be reduced by encapsulating complex notions in simple abstractions; it helps to have an overview and to see the implications of a theory. The problem here is, as Terry Winograd puts it, "one of human understanding - the ability of a person to understand how a new situation experienced in the world is related to an existing set of representations, and to possible modifications of those representations" (1990, p. 179/180).

I often start with a procedural description of a problem, that specifies how to obtain a certain result. Hereby all the knowledge was implicitly represented in the control structure (i.e. ways of processing the data) of the program. In subsequent phases the concrete and stable information would crystallize and become more and more explicit, finally moving to the data structure (i.e. the data objects themselves), reducing the amount of information that 'hides' in the control structure. For example, a list of numbers is a data structure. Its cardinality is not represented explicitly in the data structure. One can describe an algorithm that calculates its cardinality. In this case the control structure has implicit ways of determining the cardinality of lists. This information or knowledge can be made explicit by moving it from the control structure (i.e. the algorithm calculating the cardinality of a list) to the data structure, turning it into an attribute of the list itself (i.e. a number describing the cardinality of a list becomes part of the data structure). The process of understanding a problem by balancing between procedural and declarative-styled programming proved to be very fruitful.

In a lot of cases, by factorizing the procedural information and distributing it among the data components, the resulting knowledge representation was made to contain its own special built-in behaviour. Such a kind of knowledge representation can be envisioned as

a collection of small mechanical machines that have an obligatory behaviour that is completely determined by their type or character. The operations acting on these 'machines' get this behaviour for free; the machines always perform their fixed and simple routine in the appropriate place. This real integration of knowledge into a representation is thus an important aspect of the methodology.

Is a microworld more or less than a theory?

After this first exploratory phase of constructing and using a microworld, hopefully the understanding of the problem domain is improved. The next stage can then be to construct a theory. Here again a computational version of the theory in the form of a microworld has a number of advantages. After this formalisation we can recapture the implications of the theory, and in the process better understand how to achieve abstractions and true generalizations. To build and, even more important, to use such a microworld formalisation brings out aspects never foreseen during the design of a theory. It makes the theory concrete and verifiable. The construction process itself may even influence its design by revealing flaws and missing aspects. "The electronic computer gave new embodiment to mechanical rationality, making it possible to derive consequences of precisely specified rules" (Winograd, 1990, p. 169). As such, a microworld is more than a theory.

But there are also some dangerous aspects that can be associated with the construction of programs or microworlds. One frequently sees, in a computational approach to music, that a class of problems is described, followed by a description of a program and a description of the results obtained from sample problems. Often this is just one of a small set of problems with an unclear relation to the class of problems the program or the methods embody. This is the well-known "bulky program - this is the theory"- mistake. It is unclear what the limitations are, which aspects are generalizations, which aspects are specific to a particular problem, and which can be attributed to a whole class. If these limitations are not stated along with the program, the program is more or less a 'black box'; the program works in a particular case, but we don't know precisely why, and even more important, we have no idea when it doesn't work. There is a danger of starting to live in the self- created microworld, rigourosly explaining all other problems in terms of this world, instead of retaining flexibility and awareness of a certain set of un-treated problems. As such, a microworld is far from a theory.

CONCLUSION

I have argued for the importance of the use of a computational approach in music research, reaching from musicology to the psychology of music. The construction of microworlds plays a key role in different stages of the research. In this process three phases can be distinguished: the exploratory phase, where a problem is explored to gain understanding and make it concrete; secondly, a phase where a program implements the theory in a computational form that makes it explicit and open to further inquiry, and allows for tests on completeness and internal consistency; and finally a third phase where computational theories are trimmed down to their bare essence, i.e. stripped of unnecessary detail, where one is forced to make generalisations and abstractions.

This process of reducing problems to their bare essence does not come for free. The methodology does not help in taking the right decisions. So a philosophy or strategy has to be there in stepping through these phases, to help to decide what is and what isn't important. The most important characteristics of a microworld are, besides its exploratory strength - the way it makes makes abstract problems concrete, the relative ease of finding and making new abstractions and generalisations within and between related microworlds.

Finally, every theory and program will have its limitations. These should be understood and known at all times, and have to be clearly set out alongside the description of the microworld. Since they only model a very small aspect of the real world, it is important to provide all the information about how to extend and maintain them.

REFERENCES

- Abelson, H. & A. diSessa (1980) Turtle Geometry: Computation as Medium for Exploring Mathematics. Cambridge, Mass.: MIT Press.
- Desain, P. & H. Honing (1986) LOCO, Composition Microworlds in Logo. In Proceedings of the 1986 International Computer Music Conference, edited by P. Berg. San Francisco: Computer Music Association.
- Desain, P. & H. Honing (1991) The Quantization Problem: Traditional and Connectionist Approaches. In Musical Intelligence, edited by M. Balaban, K. Ebcioglu & O. Laske. Menlo Park: The AAAI Press. [forthcoming].
- Desain, P. (1990) Parsing the Parser. A Case Study in Programming Style. Computers and Music Research. Vol. II, Fall 1990.
- Dreyfus, H. (1981) From Micro-Worlds to Knowledge Representation: AI at an Impasse. In Mind Design, edited by J. Haugeland. Cambridge, Mass.: MIT Press: 161-204.
- Honing, H. (1991) Issues in the Representation of Time and Structure in Music. In Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press. [This thesis].

- Marr, D. (1985) Vision: the philosophy and the approach. In: Issues in Cognitive Modeling, edited by A. M. Aitkenhead and J. M. Slack. London: Lawrence Erlbaum Ass.
- Minsky, M. (1987) Form and Content in Computer Science. In: ACM Turing Award Lectures, edited by R. L. Ashenurst & S. Graham. Reading, MA: Addison-Wesley.
- Papert, S. (1984) Computers en Kinderen. Ontstaan en werking van de programmeertaal LOGO. Amsterdam: Bert Bakker. Originally published in 1980 as Mindstorms. New York: Basic books.
- Winograd, T. & F. Flores (1987) Understanding Computers and Cognition. A New Foundation for Design. Reading, Mass.: Addison-Wesley.
- Winograd, T. (1972) Understanding Natural Language. New York: Academic Press
- Winograd, T. (1990) Thinking machines: Can there be? Are we? In: The Foundations of Artificial Intelligence. A Source Book, edited by D. Partridge and Y. Wilks. Cambridge: Cambridge University Press.

CONCLUSION

I have argued for the importance of the use of a computational approach in music research, reaching from musicology to the psychology of music. The construction of microworlds plays a key role in different stages of the research. In this process three phases can be distinguished: the exploratory phase, where a problem is explored to gain understanding and make it concrete; secondly, a phase where a program implements the theory in a computational form that makes it explicit and open to further inquiry, and allows for tests on completeness and internal consistency; and finally a third phase where computational theories are trimmed down to their bare essence, i.e. stripped of unnecessary detail, where one is forced to make generalisations and abstractions.

This process of reducing problems to their bare essence does not come for free. The methodology does not help in taking the right decisions. So a philosophy or strategy has to be there in stepping through these phases, to help to decide what is and what isn't important. The most important characteristics of a microworld are, besides its exploratory strength - the way it makes abstract problems concrete, the relative ease of finding and making new abstractions and generalisations within and between related microworlds.

Finally, every theory and program will have its limitations. These should be understood and known at all times, and have to be clearly set out alongside the description of the microworld. Since they only model a very small aspect of the real world, it is important to provide all the information about how to extend and maintain them.

REFERENCES

- Abelson, H. & A. diSessa (1980) Turtle Geometry: Computation as Medium for Exploring Mathematics. Cambridge, Mass.: MIT Press.
- Desain, P. & H. Honing (1986) LOCO, Composition Microworlds in Logo. In Proceedings of the 1986 International Computer Music Conference, edited by P. Berg. San Francisco: Computer Music Association.
- Desain, P. & H. Honing (1991) The Quantization Problem: Traditional and Connectionist Approaches. In Musical Intelligence, edited by M. Balaban, K. Ebcioglu & O. Laske. Menlo Park: The AAAI Press. [forthcoming].
- Desain, P. (1990) Parsing the Parser. A Case Study in Programming Style. Computers and Music Research, Vol. II, Fall 1990.
- Dreyfus, H. (1981) From Micro-Worlds to Knowledge Representation: AI at an Impasse. In Mind Design, edited by J. Haugeland. Cambridge, Mass.: MIT Press: 161-204.
- Honing, H. (1991) Issues in the Representation of Time and Structure in Music. In Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press. [This thesis].

II

ISSUES

II

ISSUES

ISSUES IN THE REPRESENTATION OF TIME AND STRUCTURE IN MUSIC

Henkjan Honing

DECEMBER 1990
EDITED JULY 1991

Will be published as: Honing, H. (1991) Issues in the representation of time and structure in music. In Proceedings of the second Music and the Cognitive Sciences Conference 1990, Cambridge, edited by I. Cross and I. Deliège, *Contemporary Music Review*. London: Harwood Press.

© copyright 1990, Henkjan Honing

CONTENTS

Keywords	3
Introduction	3
Different perspectives.....	4
Music analysis and production.....	4
Musicology	4
Computer music.....	5
Music publishing and retrieval systems	5
AI and cognitive modeling	6
AI and knowledge representation	6
Cognitive and computational psychology	6
Music perception and cognition	6
General approaches to representation.....	7
Knowledge representation hypothesis	7
Procedural and declarative approaches	8
Mixed and multiple representations.....	9
Issues in music representation.....	9
The primitives: building blocks of a representation.....	10
• Decomposability	10
• Continuous or discrete?	10
The relations: issues in structuring.....	11
The representation of time.....	12
Tacit time structuring.....	12
Implicit time structuring.....	12
• Primitives: points vs intervals	12
• Time base: absolute vs relative	13
• Granularity: discrete vs continuous	13
Explicit time structuring	13
• Controversy: declarative vs procedural	14
The representation of structure.....	14
Tacit structural relations	14
Implicit structural relations.....	14
Explicit structural relations.....	15
• What kinds of structural types are needed?	15
• Relations between musical constructs: generalization vs dedication	16
• Direction: bottom-up, top-down or both?	16
• Musical structure: association with time intervals and their constraints essential	20
• Multiple representations: power vs coordination and consistency	20
• Modularization: musical knowledge vs annotation	21
Conclusion.....	21
Acknowledgements.....	23
References.....	23
Notes.....	28

ISSUES IN THE REPRESENTATION OF TIME AND STRUCTURE IN MUSIC

Henkjan Honing

*Centre for Knowledge Technology, Lange Viestraat 2b, NL-3511 BK Utrecht
Music Department, City University, Northampton Square, UK-London EC1V OHB*

This article discusses the issues in the design of a representational system for music. Following decisions as to the primitives of such a system, their time structure and general structuring is discussed. Most of the issues are presented as controversies, using extremes to clarify the underlying problems. Associating time intervals and their constraints with the components of musical structure turns out to be essential. These constraints on time intervals model an important characteristic of musical knowledge and should be part of the representation, i.e. part of the syntax. It is concluded that a representation of music should, in the short run, be made as declarative, explicit and formal as possible, while actively awaiting representation languages that can deal with the presented issues in a more flexible way.

KEYWORDS

Representational systems, music representation, knowledge representation, temporal representations, structure

INTRODUCTION

This article describes a number of important issues in the representation of music with respect to the structuring of musical information. The set of issues presented is in no way complete, but indicates the most influential decisions that have to be taken in the representation of structure. The identification of the problems is central and there will be no speculation on possible solutions. The discussion will be restricted to the *descriptive* issues of music representation, concentrating on its primitives and their structuring. Of course, a purely technical description of a representation of music is not sufficient; its cognitive aspects should be incorporated as well. Although a discussion on the modeling of the "musical mind" is not the aim here, a cognitive viewpoint will add an essential perspective in the identification of the issues in the design of a general representation of music. Since a representation of the real world (*represented* world) has to do with cognition, the image (*representing* world) will have most of cognition's characteristics.

In the cognitive sciences, and in particular subfields like computational psychology and artificial intelligence, the use of computational models (or representational systems) is central. Their merits, together with the proposal of the term "cognitive science", were described by Christopher Longuet-Higgins as:

[...] it sets new standards of precision and detail in the formulation of models of cognitive processes, these models being open to direct and immediate test. (Longuet-Higgins, 1973)

The hope is that these formulations will contribute to a new theoretical psychology. Apart from the discussion whether a computational psychology is possible at all, a computational theory sets an important foundation: by describing a theory in terms of a formal system, together with its interpretation, it can be used to define what is faulty or inadequate (i.e. it can be falsified) and might help us in defining what kind of theoretical power we actually need. Or, as Margaret Boden states:

It provides a standard of rigour and completeness to which theoretical explanations should aspire (which is not to say that a program in itself is a theory). (Boden, 1990, p. 108)

Representation is an essential part of such a formal system and decisions made in its design will undoubtedly influence the behavior of the computational model, embodying the theory. It is these decisions, to be made with regard to a representational system of music, that this article is aiming at.

DIFFERENT PERSPECTIVES

A number of different areas of research have a direct interest in specifying an appropriate representation of music. The latter either forms the basis of their studies or is a subject of study in itself. In the following short overview the different viewpoints and their specific demands will be described. The main difference is contained in the distinction between representations of a technical nature and representations of a cognitive nature (conceptual or mental representations).

Music analysis and production

Musicology

Notation has always played a central role in musicological research. The design and adaptation of notations or representations have been developed along with the specific theories of analysis. Different overlapping or contradicting theories have been proposed (Schenker, 1956; Meyer, 1973; Narmour, 1977; Lerdahl & Jackendoff, 1983). Most theories agree that there is more in music than what is written in the score. In this sense, the

opinion of the philosopher Nelson Goodman (1968) that a piece can be characterized as the set of performances in conformance with its score is an exception. The question here is whether a piece of music resides in the notation, in the air, or in people's minds, or in other words, whether music is cognitive or not.

Computer music

In the field of computer music there is an interest in the design of appropriate data structures for music systems that form the basis of , for example, composition tools, interactive systems, and notation systems. Several projects have proposed different kinds of representation, suited to the specific demands of the particular problem or even to the software or hardware used (see Loy, 1988 for an elaborate survey of computer music systems). A distinction can be made between representations designed for real-time systems that are process-oriented (e.g. Puckette, 1988), and non-real-time systems that have a static global view of the music (e.g. Dannenberg, 1989). They differ, respectively, in their tacit and explicit representation of time (see below: The representation of time).

All systems have their own way of representing music and share little common ground. The only widespread standard is the industry proposed MIDI standard: a communication protocol (described in Loy, 1985) and file format. It is a very low-level stream-like and structureless representation (criticized in Moore, 1988) designed for communication between electronic instruments and computers. Within the computer music community several initiatives (Dannenberg et al, 1989; ANSI, 1989) have been taken towards a more general and high-level representational standard.

Music publishing and retrieval systems

In music archiving the need for the standardization of notated music has resulted in several proposals for the storage and printing of music (Erickson, 1975; Byrd, 1984; Gourlay, 1986). Most of them are based on a visual description (e.g. notes positioned on staves) and are not very general in their applicability. The ANSI standardization committee for music representation (ANSI, 1989) is a recent attempt to make a technical and methodological specification for a standard music description language, useful in areas such as music publishing, music databases, computer assisted instruction, music analysis, and music production. In general, these standards seem to concentrate more on pragmatics (e.g. efficiency, in terms of size and speed requirements) than on generality and consistency.

AI and cognitive modeling

Another large area of research is artificial intelligence (AI) and the cognitive sciences. Both have their own specific goals and demands. I will describe them here briefly.

AI and knowledge representation

In AI the concern is to notate descriptions of the world in such a way that an intelligent machine can come to conclusions about its environment by formally manipulating these descriptions. In knowledge representation, a subfield of AI, research is focussed on the development of representation languages and the design of inference schemes (e.g. to model reasoning about knowledge). Both are based in the tradition of (predicate) logic while more recent languages can be classified as structured object representations (e.g. frames; Minsky, 1975), associational representations (e.g. semantic networks; Quillian, 1968), and procedural representations and production systems (Newell, 1973). It is important to note that AI and knowledge representation are about feasible ways to build intelligent systems and not so much about modeling cognitive behavior.

AI and music is also an important field of research where representation is becoming one of the central issues (Balaban et al., 1991).

Cognitive and computational psychology

In the cognitive sciences, mental and knowledge representations are important subjects of study. It seems impossible to imagine a cognitive system in which a representation does not play a central role (Anderson, 1983; Fodor, 1983; Johnson-Laird, 1983). There is, however, no general agreement on the assumption that mental activity is mediated by internal or mental representations, and when there is, there is still some discord on the precise nature of these representations. Proposals for knowledge representation can be grouped into three categories: propositional representations (discrete symbols or propositions), analogical representations (use of images), and procedural representations (i.e. modeled as processes or procedures). To this last category also belong distributed representations (e.g. connectionist networks).

Music perception and cognition

In the psychology of music, alongside research in music production and comprehension, the majority of work has consisted of describing the nature of musical knowledge and its representation. Elaborate studies have been done in the domains of pitch (Krumhansl, 1979; Shepard, 1982), rhythm (Povel & Essens, 1981; Longuet-Higgins & Lee, 1984; Desain & Honing, 1989) and timbre (Grey, 1977; Wessel, 1979). But here also, there is no general agreement on the precise nature of these representations (see McAdams, 1987 for a more complete overview or Sloboda, 1985; Dowling & Harwood, 1986).

GENERAL APPROACHES TO REPRESENTATION

This paragraph will outline the main approaches to representation. Identifying the problems of representation in general will be shown to be of direct benefit to the debate concerning music representation.

Knowledge representation hypothesis

An important assumption in a formalist approach to representation is the *knowledge representation hypothesis*. It is summarized by Brian C. Smith (1982) as follows:

Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantical attribution play a formal but causal and essential role in engendering the behavior that manifests that knowledge.

Such a “mechanically embodied intelligent process” is presumed to be an internal process that manipulates a set of representational structures, in such way that the intelligent behavior of the whole results from the interaction of parts. It is presumed only to react to the form or shape of these representations, without regard to what they mean or represent.

As an illustrative example one can use a technique that is sometimes used in making enlarged copies of pictures, for instance, by artists who make large chalk drawings of well-known paintings on the street. They copy these paintings from a small reproduction, holding the it *upside-down*. This minimizes the distorting influence a perspective has on the copying of the actual proportions: an unwanted interpretation that imposes ‘meaning’ not present in the picture. This example shows that one has to watch out for interpretive knowledge, so easily added by human observers, not present in the representation itself. A representation is only syntax and should have all knowledge embodied in this syntax, independent of the interpretive system.

A representational system can be defined as “a formal system for making explicit certain entities or types of information, together with a specification of how the system does this” (Marr, 1982). In the formalist definition entities in a formal system might have complex mechanisms.¹ In deciding on any particular representational system and its entities, there is a trade-off; certain information will become explicit at the expense of other information being pushed into the background making it possibly hard to recover.

Procedural and declarative approaches

There is a classic distinction between declarative and procedural ways of representing knowledge: declarative being the knowledge *about* something, while procedural knowledge states the knowledge in terms of *how to do* something. Declarative knowledge tends to be accessible: it can easily be examined and combined. Procedural knowledge tends to be inaccessible, guiding a series of actions but allowing little examination. We seem to have conscious access to declarative knowledge whereas we do not have this access to procedural knowledge (Rumelhart & Norman, 1985).

Declarative representations have the merit of being composable i.e. the meaning of a complex expression is based on or can be derived from the meaning of its parts and their combinations. There are no interactions between separate entities, which makes the representation extremely modular. Knowledge can simply be added as long as it keeps the system consistent. All knowledge is open for introspection.

In procedural representations the emphasis is on interaction. Procedural representations are, not surprisingly, very powerful in modeling knowledge that is procedural by nature. There is no separation between facts and processes. Interactions are strong but deriving semantics is very hard (if not impossible). Addition or change is only reached by modification (and a resulting debugging process). Introspection and reflection is impossible. The problem, here, is the way in which procedures can be represented so that they can be interpreted. The question becomes *what* they do, instead of *how* they do it (see Table 1 for an overview).

Declarative knowledge	Procedural knowledge
accessible	inaccessible
modular (no interaction)	interaction (no separation between facts and processes)
composable semantics	impossible (or hard) to derive semantics
open to introspection and reflection	closed to introspection and reflection
knowledge can easily be added, if consistent	addition only by modification
control structure obscure	control structure explicit

Table 1. Procedural and declarative knowledge representations compared.

Mixed and multiple representations

In general, the distinctions between procedural and declarative representations are about efficiency, control, modularity, and the accessibility of knowledge. For computer science the first two are most important, while cognitive psychology is most interested in the last two.

Terry Winograd (1975) emphasized the duality between modularity and interaction, interaction being a strong characteristic of procedural representations and modularity of declarative representations. Many complex systems can be viewed as “nearly decomposable systems”, a notion introduced by Herbert Simon (1969).² A single module can be studied separately without constant attention to its interaction(s) with other modules. Interactions among these subsystems are weak but not negligible. In representational terms, this forces us to have representations that facilitate these weak interactions. *Mixed* representations (i.e. both modular and interactive), as described by Winograd and others, have been further developed in the design of object-oriented languages (e.g. Minsky, 1975; Hewitt, 1975). In *mixed* representations different *parts* of the represented world are described in different ways. Some parts might be described procedurally, while others are described in a declarative way.

Another approach is to have *multiple* representations of the same ‘world’, each describing the represented world *completely*. Instead of a mixture of, for example, procedural and declarative representations, describing different parts of the world, there is a procedural representation describing the whole world and a declarative representation describing the whole world in parallel. Here the trade-off is extra power against the problem of coordinating the information in the separate representations: when a change is made, all structures have to be kept consistent so as to reflect the same represented world.

ISSUES IN MUSIC REPRESENTATION

The remainder of this article will address issues specific to the representation of music. Three sub-areas will be discussed: the primitives of a music representation, time structuring and general structuring. The notion of structuring depends on the possibility of decomposing a representation into meaningful entities, so we must first answer the important question: *what* are we structuring?

The primitives: building blocks of a representation

• Decomposability

How to decompose a representation of music into the appropriate parts? What are the building blocks, the primitives of such a representation? As described earlier, this decision is essential and has implications on what kind of information will be lost and what information will clearly be represented.

There seems to be a general consensus on the notion of discrete elements (e.g. notes, sound events or objects) as the primitives of music. It forms the basis of a vast amount of music-theoretical work and research in the psychology of music, but a detailed discussion and argument for this assumption is missing from the literature. In music theory, as Robert Erickson (1982, p. 533) points out, there is no clear definition of what such a primitive object might be. In the psychology of music, John Sloboda (1985, p. 24), for example, just states "the basis phoneme of music is a note", and Diana Deutsch (1982) founds her discussion on grouping mechanisms in music on a 'given' set of basic acoustic elements. Yet the essential question of what these elements or 'phonemes' are is not answered. Research in psycho-acoustics on streaming shows how difficult it is to decide on such elements from a perceptual point of view (McAdams & Bregman, 1979; Bregman, 1990). A distinction has to be made between natural and artificial discretization of dimensions, or, in other words, the existence of possibly innate perceptual mechanisms and a learned division of continuous signals. In going from a continuous acoustic signal to a discrete signal one loses information. This quantization process should be looked at as a separation process instead: both types of information, the continuous and the discrete, are needed, and probably interact with each other (cf. Desain & Honing, 1989, with regard to this separation process in rhythm perception). So, next to decomposition, the issue of the characterization of the primitives of a representation, as continuous, discrete or a combination of the two, is very important.

• Continuous or discrete?

By way of illustration, imagine Billie Holiday singing "*I cried for you.*" How can the sound be represented in such a way that all expressive and structural information is incorporated? What is the relation between the actual perception and the notes originally notated in the score? Consists the sentence as sung of several discrete entities, or should it be described in a continuous way? Or a combination of both? For example, discrete phonemes, syllables or notes, continuous expression *over* these discrete structural elements, continuous fluctuations of pitch and amplitude *within* them, etc. combined into several levels of discrete and continuous types of information that are closely related.

In music cognition, the assumption of discrete elements finds a lot of support (McAdams, 1989). Stephen McAdams makes a distinction between three auditory grouping processes that organize the acoustic surface into musical events, connect events into musical streams, and 'chunk' event streams into musical units (simultaneous, sequential and segmentational grouping, respectively); and perceived discrete qualities that are based on learning (e.g. scale, meter, harmony) (McAdams, 1989, p. 182). These discrete elements of music are assumed to carry structure, while the continuous aspects carry expression (Clarke, 1987). Mary Louise Serafine (1988) stands quite alone in arguing for a continuous basis. She blames music perception research for reducing music to false elements such as discrete pitches, scales and chords: "[they] are not the elements or building blocks of music" (p. 52). She accounts for these elements as an after-the-fact notion of music. But, as David Huron (1990) observes, these are speculative claims with no empirical support. It is clear that there is still quite a lot of discussion and research needed, especially on the rules of the segregation of acoustic signals, before we can decide on the discrete elements of a general representation of music.

Currently, most music representation systems use either notes or sound events/objects as the building blocks of their descriptions. In these systems, the distinction between continuous and discrete is normally between sound generation and the discrete events which describe the sound in several attributes, or, in other words, between the instrument and the score. This division rests on the assumption that sound is continuous by nature (e.g. signals, wave forms), whereas the score is mainly a collection of discrete events. The continuous aspects of the score (e.g. timing and dynamics) are often taken care of by different kinds of procedures or 'modifiers' (e.g. Pope, 1989; Dyer, 1990) acting *on* the score: their descriptions are not part of the score representation (see below: Granularity). The trade-off made in these decompositions is very little discussed or even acknowledged.

The relations: issues in structuring

When we have decided on the primitives of the representation, their structuring becomes of great importance. This structuring will be described in two separate sections. Since time and its structuring is an important factor in music, with its own specific issues related to it, it will be discussed separately from the issues in general structuring. However, in the end it will be shown that they are not very different. Time structuring will be discussed first.

THE REPRESENTATION OF TIME

A number of distinctions need to be made in trying to narrow down discussion of the representation of time. There are three different areas of interest: temporal representation, temporal logic or reasoning, and planning and scheduling. All of them influence the design of a representation of time. This section will concentrate on the first.

The representation of time can be subdivided in three categories: 1) *tacit* (time is not represented at all); 2) *implicit* (time is represented, but explicit time relations are not); and 3) *explicit* (time is represented with explicit time relations). The issues will be spread over these categories.

Tacit time structuring

Some real-time systems can be called 'no-time' systems (e.g. Bharucha, 1987; Puckette, 1988). Because time is not explicitly represented in the primitives, there is only the notion of *now*. There is no explicit formulation of the systems dependence on time and no information regarding time (except 'now') can be derived or manipulated.

Implicit time structuring

In this category, time is represented without explicit time relations. Time is expressed in an absolute way (e.g. note lists (Matthews, 1969)) or relative to an arbitrary point of reference. Time relations (e.g. this note occurs before that note, or, these notes are overlapping) have to be calculated since they are not explicitly stated in the representation.

• Primitives: points vs intervals

The decision to represent time as points or intervals is not arbitrary, even when they, theoretically, can be expressed in terms of each other (an interval is a collection of points, a point is a very short interval³). A point-based representation (McDermott, 1982) implies the occurrence of only one event at a time and lacks the concept of an event 'taking' time. As Allen (1983) argues, there seems to be a strong intuition that, given an event, we can always "turn up the magnification" and look at its structure. He therefore proposes an interval-based representation. Intervals form a strong basis for the computability of meaningful relations, i.e. time intervals that overlap, meet, are during, before, and after each other, etc.

In music representation there are examples of both choices. Mira Balaban (1989), for instance, describes a representation based on pairs of a sound object and a time point, and

Desain & Honing (1988) use sound objects with a duration (i.e. time interval) as the basis of a representation of time.

- **Time base: absolute vs relative**

The time base that can be chosen is either absolute or relative, or, in other words, real-time (e.g. in seconds) or proportional time (e.g. a quarter note). With an absolute time base, (onset-)time is an attribute of the musical object, whereas with a relative time base it isn't.

Some music representation systems (Smith, 1972; Schottstaedt, 1983) use lists of notes with absolute times, whereas later systems tend to describe time in terms of a relative time base or relative to the enclosing time context, i.e. expressed as a function of this context (Dannenberg, 1989; Balaban, 1989). But both time bases seem to be needed. For example, in representing a trill as being twice as long as another trill, one has to decide whether to stretch or to extend the description of this related trill, i.e. is the new trill half the speed (using relative time) or is the speed the same (using absolute time) and are there just more notes added (or any other particular way of extending a trill). Both types of behavior, using both time bases, need to be represented to allow for both representations of time.

- **Granularity: discrete vs continuous**

What is the grain or grid size of the time bases mentioned above? Is time expressed as a discrete value labeling events, or is it expressed as a continuous function? As well as discrete time, a continuous way of representing time is needed, for example, when representing an *accelerando* or *rubato* over a series of notes.⁴ Most representational systems make these notions available as global operations acting *upon* the representation instead of making them *part* of the representation.

Explicit time structuring

An example of explicit time structuring in music is the use of two basic structuring relations called 'parallel' and 'sequential' (Desain & Honing, 1988). These two time relations, and combinations of them, can express many constellations of discrete sound events. Similar time structuring is proposed by several other authors (e.g. Rodet & Cointe, 1984; Dannenberg, 1989). Allen (1983) describes a list of thirteen possible relationships. A set of basic explicit time relations forms a solid basis for higher level notions of time structuring and make operations on time, or operations depending on time, very elegant (Desain, 1990).

- **Controversy: declarative vs procedural**

The controversy over declarative and procedural representations is also very important in the representation of music. Take the example of a trill - a sequence of notes, alternating in pitch, filling up a certain time interval. This "filling up" is most naturally represented in a procedural form. But, as discussed previously, this type of representation has quite some disadvantages. Problems occur when there is, for instance, a nesting of these trills defined in terms of each other (e.g. a higher-level trill composed by combining the definitions of some other, i.e. lower-level trills): the definition of the high-level trill depends on the *result* of the low-level trills, a result that is only available *after* execution of the procedural description of these low-level trills. There is no way in which the duration of the high-level trill can be decided upon without evaluating the definition of the low-level trills since this knowledge is represented in a procedural form. The declarative representation (a low-level trill of a certain length) has to be replaced by the result (a sequence of notes adding up to a certain length) and information is lost (e.g. knowledge on how the trill was composed). Both kinds of representation seem to be needed in the representation of music. The marriage of both types of knowledge is, as described before, still a topic of research.

THE REPRESENTATION OF STRUCTURE

Structural descriptions of music can be divided into two areas. One is the description of musical structure independent of psychological considerations, based on an analysis by a musicologist. The other is the description of the structural properties of mental representations of music: the goal of music psychology research. The described issues are relevant to both areas. In describing general structuring, we can employ the same division used in the subfield of time structuring: 1) *tacit* structural relations, 2) *implicit* structural relations, and 3) *explicit* structural relations.

Tacit structural relations

When no structure is represented, we are left with only the primitives of the representation. This is the case in the earlier mentioned MIDI protocol that represents a piece of music as a structureless stream of note-onsets and offsets (with as attributes an integer key number, a velocity value and channel number).

Implicit structural relations

Implicit are those structural relations that have to be calculated from the representation. As an example, from a MIDI file format the following structural information can be obtained: all notes on channel 1 belong to one unit called a 'track';

every two seconds there is a bar and all notes within that time span are part of it; etc. The structural relations that can be derived from a representation (with only implicit structuring) depend heavily on the choice of primitives and their attributes.

Explicit structural relations

Structure is the denominator for a large class of possible relations made between the entities of a representation. One can say that almost everything, except the entities themselves, is structure. Very few representational systems for music supply explicit structuring mechanisms, and even when they are available, they only represent specific kinds of structure (e.g. meter, bars, instrumental parts) or support annotation (e.g. "this is an important note"). The following paragraphs discuss the issues in the design of a general structuring mechanism.

• What kinds of structural types are needed?

One way of describing different kinds of relations -so as to have a handle to talk about them in a general way- is to divide them in binary and n-ary relations. A special kind of binary relation is a tree or hierarchy. A *part-of* relation defines such a hierarchical relation between objects. It propagates behavior between objects. A part-of relation could denote relations such as "all notes part-of chord", or the often-used bar, beat, and note hierarchy. They are quite general and flexible in describing musical structure (see Honing, 1990).

Another hierarchical relation, orthogonal to the part-of relation, is the *is-a* relation. It defines inheritance of behavior and characteristics, specifying a generalization hierarchy of objects: a structure of concepts which are linked to those of which they are specializations. Examples are: a dominant chord being a special kind of seventh chord, a chord being a kind of cluster, a cluster being a kind of collection of notes, etc. (see e.g. Pope, 1989).

A great number of music theories use hierarchies as their only kind of structuring (Lerdahl & Jackendoff, 1983). Hierarchies are very useful in relating local and global information, but other kinds of relations are needed as well. Other binary relations like associative relations are useful in relating, for example, a theme with its variations. Functional relations are also needed (e.g. the function of a particular chord in a scale) as well as referential relations (e.g. a theme referring to a previously presented or already known motif).

N-ary relations can structure more complex types of relation: for instance, the dependency of a certain chord on scale, mode and the context in which it is used is a ternary relation.

The structural types described here are the ones most relevant to music, though a complete overview of all musical constructs and their expression in these structural types would take considerably more space.⁵

- **Relations between musical constructs: generalization vs dedication**

Not everything is said about musical structure by simply assigning one of the structural types described above. Within one type of structure (e.g. defined in terms of part-of relations) refinement is needed to distinguish between the different musical constructs described by means of this type (e.g. what is the difference between a chord and a bar when both are described in terms of part-of relations?). There are two extremes in approaching this problem. One approach is dedication: all the well-known or often used musical constructs (chord, arpeggio, bar, beat, trill, grace note, etc.) are described, more or less ad hoc, as primitives with their own specific relations (and resulting behavior), with little or no hierarchy. The other approach is generalization and is based on parsimony: there are no special musical constructs defined as primitives, all constructs being based on some very general primitive (e.g. a time interval). The bias is on generality: new musical constructs have to be defined in terms of existing ones, in a hierarchical way.

The first is a popular and pragmatic approach. For instance, in a computer composition system a reasonable set can be provided that takes care of most needs. The main drawback is that extensions have to be made in an ad hoc fashion and often need to have their own processes (or transformations) defined for the user to be able to access or manipulate them.

In the latter approach the choice of the right generalities is the problem. But when they are available, extensions are simply defined in terms of these generalities or higher-level constructs. There is no need to 'tell' the processes, acting on the representation, about these new constructs.

- **Direction: bottom-up, top-down or both?**

In expressing one of the above mentioned relations, it is important to note how the information flow is supported by the representation. In music theory and the psychology

of music, different directions are proposed: from the conceptual level down (top-down; Schenker, 1956), and from the low-level data up (bottom-up; Narmour, 1977), or in both directions, as in modeling tonal hierarchies with interactive activation networks (Bharucha, 1987).

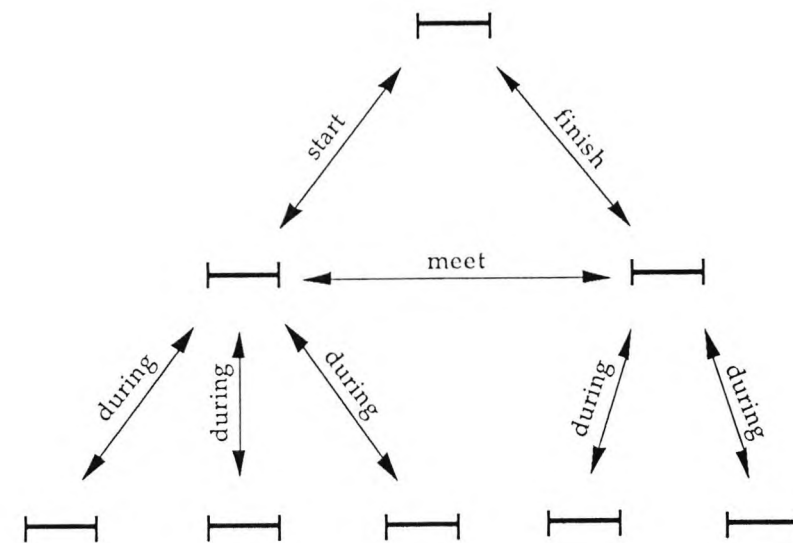
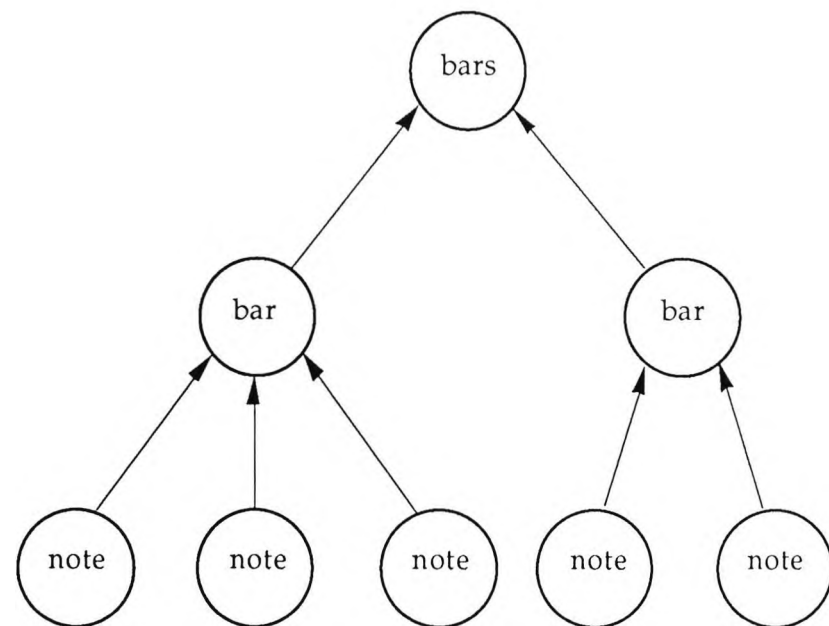


Figure 1. A 'bars' structure with part-of relations (a), its associated time intervals and constraints (b).

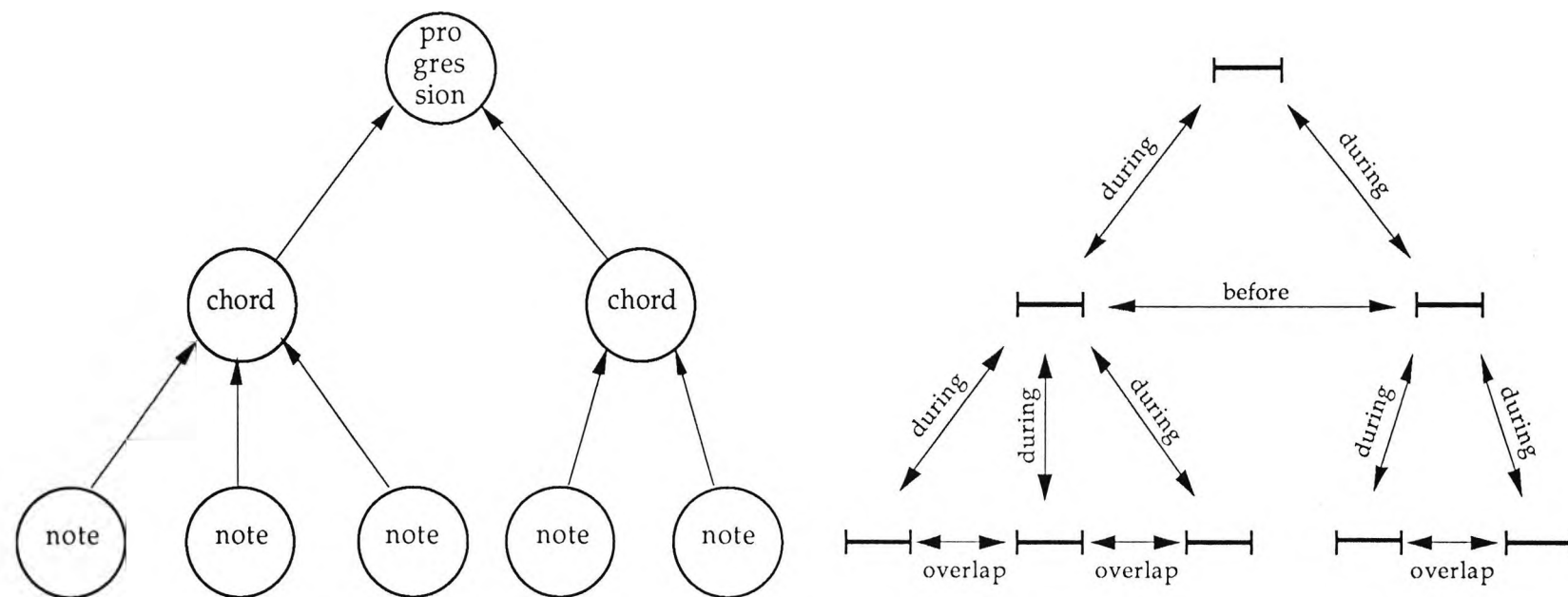


Figure 2. A 'progression' structure with part-of relations (a), its associated time intervals and constraints (b).

- **Musical structure: association with time intervals and their constraints essential**

In what way is musical structure different from any general structure mechanism (e.g. the part-of and is-a relations we described before)? Since time is an influential factor in most, if not all types of structure in music, musical structure can be described as a collection of structuring mechanisms that have time intervals associated with their components (i.e. structural objects). It is the constraints on these time intervals that specialize the different types of structuring.

As an example, let's look at two simple part-of relations: bars, a bar, note (see Figure 1a), and a progression, chord, note hierarchy (see Figure 2a). In the first hierarchy it is clear that the structural object 'bars' and its parts have a duration: they hold for a certain time interval. This is also the case for the 'progression' object and its parts. Both constructs have the same part-of structure but differ in the kind of constraints they have on their associated time intervals. In a 'bars' structure, if one bar becomes longer, the other one has to become shorter: they have to satisfy the *meet* constraint (using Allen's (1983) terminology). In the 'progression' structure, the comparable structural objects have a *before* relation. The musical constructs are characterized by the specific constraints on these time intervals associated with their structural objects (see Figure 1b and 2b)⁶.

These constraints should be part of the representation, i.e. part of the syntax, so that operations on the representation produce the behavior resulting from these restrictions for free; the semantics of musical constructs (e.g. what does an arpeggio mean, and how does it differ from a chord or a run of notes) should be moved to the syntax. In this way the representation has embedded knowledge of how to deal with particular kinds of structure. These musical constructs can be compared with small machines: they have a clear and accessible behavior that cannot be altered.

- **Multiple representations: power vs coordination and consistency**

Multiple representations are needed in a complete description of music, i.e. several structural descriptions being applied to the same primitives (e.g. a note is part of a meter and a tonal hierarchy at the same time). One could think of multiple structural representations as analogous to a ring binder: the spiral resembles the primitives, the pages the different kinds of structural relations.⁷ As described before (see General approaches to representation), the consistency and coordination of the information between the pages is the problem here.

Inconsistencies may occur when two structural descriptions clash (i.e. the constraints on both structural descriptions can't be solved or unified) and exceptional or preferred behavior has to be provided. It seems that in these situations, the demand for consistency is too strong (e.g. a slowed-down chord structure might turn into an arpeggio). It may not be possible to formalize a representation of music in a way that guarantees consistency.⁸ More research is needed in the formalization of musical constructs (i.e. definition and behavior) and their combination that might result in exceptional or preferred behavior.

- **Modularization: musical knowledge vs annotation**

Here the issue is whether structuring is used to add musical knowledge or just used as annotation. Structure can be used as an annotation of the basic elements of the representation assigning different kinds of information, but it can also be interpreted as musical knowledge. Using structure in both ways facilitates modularity: not all knowledge about music has to be part of the representation, since structure can be used as a hook to import information from outside the system. This improves the modularity of the system considerably (as advocated by Simon (1969) in technical terms, and by Fodor (1983) in cognitive terms).

CONCLUSION

Representational systems have a central position in the cognitive sciences, especially in the fields of computational psychology and artificial intelligence. A formalist approach to representation, as summarized in the "knowledge representation hypothesis", applied to the representation of music has turned out to be beneficial. Representing musical knowledge in syntactical terms, makes a theory within the psychology of music explicit and verifiable. Discussing the issues in the design of such a representational system for music is what this article has aimed at.

Before talking about structuring, the question "what are we structuring?" was asked. The decomposability of a representation of music was discussed as well as the expression of its primitives in either discrete or continuous terms (or a combination thereof). Research in the segregation of acoustical signals (Bregman, 1990) is essential in deciding on the primitives of a *general* representation of music. Currently, most research is based on the assumption that the basic elements of music are discrete.

The discussion of time structuring, as a special case of general structuring, showed that the choice of either points or intervals, a relative or absolute time base, discrete or

continuous representations, and the use of procedural or declarative descriptions of musical knowledge are controversies where solutions through combining these polarities have to be found.

Several types of general structuring were discussed. An important point is the observation that structure in music is often associated with a time interval (for which it 'holds'). The constraints on these time intervals model specific musical constructs and their behavior. Time structuring and general structuring differ in the sense that time structuring makes these constraints explicit: they are represented as structural objects (e.g. 'parallel' and 'sequential' relations), while in general structuring they are implicit: they are used to restrict the behavior of the specific structure, but are not explicitly represented as structural objects.

In conclusion:

1) A representation should be as *formal* as possible. Even when the meaning is removed from the formal system it must be possible to prove its correctness (i.e. not dependent on knowledge outside the formal definition).

2) A representation should be as *declarative* as possible. Declarative representations were shown to have preference over procedural representations, even though some information is more naturally represented in a procedural way.

3) A representation should be as *explicit* as possible. All relations and knowledge should be explicitly stated in the representation.

4) All the controversies presented above need combined solutions in which both extremes can be expressed. The idea of having *multiple* representations of the same 'world' seems useful.

5) Musical structure should be associated with time intervals. Constraints on these time intervals model the specific musical constructs and their behavior. These constraints should be part of the representation, i.e. part of the syntax, so that operations on the representation get the behavior resulting from these restrictions for free.

In the short term, it is concluded that it would be best to construct representations of music so as to be as declarative, explicit and formal as possible, while actively awaiting

developments in representation languages or schemes that can deal with the issues presented here in a more flexible way.⁹

ACKNOWLEDGEMENTS

Thanks to David Huron, Christopher Longuet-Higgins, Steve McAdams, Stephen Pope, Maria Ramos, and my colleagues at City University, Music Department and the Centre for Knowledge Technology for useful discussions and advice. Special support by Johan den Biggelaar, Ton Hokken and Thera Jonker is highly appreciated. Thanks for proof-reading and valuable suggestions and improvements on earlier versions to Eric Clarke, Joop Ringelberg, and an anonymous referee. The research was in part supported by an ESRC grant under number A413254004. Finally, special thanks to Peter Desain for his encouragement, insights, and generous sharing of ideas.

REFERENCES

- Allen, J.F. (1983) Maintaining Knowledge about Temporal Intervals. In: Communications of the ACM, 26(11).
- Anderson, J.R. (1983) The Architecture of Cognition. Cambridge, Mass.: Harvard University Press.
- ANSI (American National Standards Institute) (1989) X3V1.8M/SD-6 Journal of Development Standard Music Description Language (SMDL). San Francisco: Computer Music Association.
- Balaban, M. (1989) Music Structures: A Temporal-Hierarchical Representation for Music. Musikometrika, Vol. 2.
- Balaban, M., K. Ebcioglu & O. Laske, eds (1991) Musical Intelligence. Menlo Park: The AAAI Press. (forthcoming).
- Bharucha, J.J. (1987) MUSACT: A Connectionist Model of Musical Harmony. In Proceedings of the Cognitive Science Society. Hillsdale, New Jersey: Erlbaum.
- Boden, M. A. (1990) Has AI helped psychology? In: The foundations of artificial intelligence. A source book, edited by D. Partridge and Y. Wilks. Cambridge: Cambridge University Press.
- Bregman, A.S. (1990) Auditory Scene Analysis: The Perceptual Organization of Sound. Cambridge, Mass.: Bradford books, MIT Press.
- Byrd, D. (1984) Music Notation by Computer. Ph. D. Dissertation, Computer Science Department, Indiana University. Ann Harbor: University Microfilms.

- Clarke, E.F. (1987) Levels of structure in the organisation of musical time. In "Music and psychology: a mutual regard", edited by S. McAdams. Contemporary Music Review, 2(1).
- Clarke, E.F. (1988) Generative principles in music performance. In Generative processes in music. The psychology of performance, improvisation and composition, edited by J. A. Sloboda. Oxford: Science Publications.
- Dannenberg, R. (1989) The Canon Score Language. Computer Music Journal 13(1).
- Dannenberg, R., L.M. Dyer, G.E. Garnett, S.T. Pope, & C. Roads (1989) Position papers. In Proceedings of the 1989 International Computer Music Conference. San Francisco: Computer Music Association.
- Desain, P. & H. Honing (1988) LOCO: A Composition Microworld in Logo. Computer Music Journal 12(3).
- Desain, P. & H. Honing. (1989) Quantization of Musical Time: A Connectionist Approach. Computer Music Journal 13(3). Reprinted and updated in Todd & Loy (1991).
- Desain, P. & H. Honing. (1991a). Tempo curves considered harmful. In "Music and Time", edited by J. D. Kramer. Contemporary Music Review. London: Harwood Press. (forthcoming).
- Desain, P. & H. Honing. (1991b). Towards a calculus for expressive timing in music. Research Report. Utrecht: Centre for Knowledge Technology.
- Desain, P. & H. Honing. (in press) Time functions function best as functions of multiple times. To appear in Computer Music Journal.
- Desain, P. (1990) Lisp as a second Language. Perspectives of New Music, 28(1).
- Deutsch, D. (1982) Grouping Mechanisms in Music. In The Psychology of Music, edited by D. Deutsch. New York: Academic Press.
- Dowling, W.J. & D. Harwood. (1986) Music Cognition. New York: Academic Press.
- Dyer, L. (1990) Ensemble. Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association.
- Erickson, R. (1975) The DARMS Project: A Status Report. Computers and the Humanities, 9(6).
- Erickson, R. (1982) New Music and Psychology. In The Psychology of Music, edited by D. Deutsch. New York: Academic Press.
- Fodor, J. (1983) The Modularity of the Mind: An Essay on Faculty Psychology. Cambridge, Mass.: Bradford Books, MIT Press
- Goodman, N. (1968) The Languages of Art: An Approach to a Theory of Symbols. Indianapolis: Bobbs-Merill Co.

- Gourlay, J.S. (1986) A Language for Music Printing. Communications of the ACM, 29(5).
- Grey, J.M. (1977) Multidimensional Perceptual Scaling of Musical Timbres. Journal of the Acoustical Society of America, 61.
- Hewitt, C. (1975) How to use what you know. Proceedings of the Fourth International Joint Conference on Artificial Intelligence. Los Altos, CA.: Morgan Kaufmann.
- Honing, H. (1990) POCO: An Environment for Analysing, Modifying, and Generating Expression in Music. Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association.
- Huron, D. (1990) Book review of Music as Cognition by M.L. Serafine. Psychology of Music, 18.
- Johnson-Laird, P.N. (1983) Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness. Cambridge, Mass.: Harvard University Press.
- Krumhansl, C.L. (1979) The Psychological Representation of Musical Pitch in a Tonal Context. Cognitive Psychology, 11.
- Lerdahl, F. & R. Jackendoff (1983) A Generative Theory of Tonal Music. Cambridge, Mass.: MIT Press.
- Longuet-Higgins, H.C (1973) Comments of the Lighthill Report. Artificial Intelligence - A Paper Symposium. London: Science Research Council. Reprinted in Longuet-Higgins (1987).
- Longuet-Higgins, H.C (1987) Mental Processes. Cambridge, Mass.: MIT Press.
- Longuet-Higgins, H.C. & C.S. Lee (1984) The Rhythmic Interpretation of Monophonic Music. Music Perception, 1. Reprinted in Longuet-Higgins (1987).
- Loy, G. (1985) Musicians Make a Standard: The MIDI Phenomenon. Computer Music Journal 9(4). Reprinted in Roads (1989).
- Loy, G. (1988) Composing with Computers - A Survey of Some Compositional Formalisms and Music Programming Languages. In Current Directions in Computer Music Research, edited by M. V. Matthews & J. R. Pierce. Cambridge, Mass.: MIT Press.
- Marr, D. (1982) Vision: A Computational Investigation into Human Representation and Processing of Visual Information. San Francisco: W.H.Freeman.
- Matthews, M.V. (1969) The Technology of Computer Music. Cambridge, Mass: MIT Press.
- McAdams, S & A. Bregman (1979) Hearing Musical Streams. Computer Music Journal 3(4). Reprinted in Roads & Strawn (1985).
- McAdams, S. (1987) Music: A Science of the Mind? In "Music and Psychology: A Mutual Regard", edited by S. McAdams. Contemporary Music Review, 2(1).

- McAdams, S. (1989) Psychological constraints on form-bearing dimensions in music. In "Music and the cognitive sciences", edited by S. McAdams and I. Deliège. Contemporary Music Review, 4(1).
- McDermott, D.V. (1982) A Temporal Logic for Reasoning about Processes and Plans. Cognitive Science, 6.
- Meyer, L.B. (1973) Explaining Music: Essays and Explorations. Berkeley: University of California Press.
- Minsky, M. (1975) A Framework for Representing Knowledge. In The Psychology of Computer Vision, edited by P. Winston. New York: McGraw-Hill.
- Moore, F.R. (1988) The Dysfunctions of MIDI. Computer Music Journal 12(1).
- Narmour, E. (1977) Beyond Schenkerism: The need for Alternatives in Music Analysis. Chicago: University of Chicago Press.
- Newell, A. (1973) Productions systems: models of control structures. In Visual Information Processing, edited by W.G. Chase. New York: Academic Press.
- Pope, S.T. (1989) Modeling Musical Structures as EventGenerators. Proceedings of the 1989 International Computer Music Conference. San Francisco: Computer Music Association.
- Povel, D.J. & P. Essens (1981) Perception of temporal patterns. Music Perception, 2
- Puckette, M. (1988) The Patcher. In Proceedings of the 1988 International Computer Music Conference. San Francisco: Computer Music Association.
- Quillian, M.R. (1968) Semantic Memory. In Semantic Information Processing, edited by M.L. Minsky. Cambridge, Mass: MIT Press.
- Roads, C (ed.) (1989) The Music Machine. Cambridge, Mass.: MIT Press.
- Roads, C. & J. Strawn (eds.) (1985) Foundations of Computer Music. Cambridge, Mass.: MIT Press.
- Rodet, X. and P. Cointe. (1984) FORMES: Composition and Scheduling of Processes. Computer Music Journal 8(3). Reprinted in Roads (1989).
- Rumelhart, D.E. & D.A. Norman. (1985) Representation of Knowledge. In Issues in Cognitive Modeling, edited by A. M. Aitkenhead and J. M. Slack. London: Lawrence Erlbaum Ass.
- Schenker, H. (1956) Der Freie Satz. Vienna: Universal Edition
- Schottstaedt, W. (1983) PLA: A Composer's Idea of a Language. Computer Music Journal 7(1). Reprinted in Roads (1989).
- Serafine, M.L. (1988) Music as Cognition: The Development of Thought in Sound. New York: Columbia University Press.

- Shepard, R.N. (1982) Structural approximations of musical pitch. In The Psychology of Music, edited by D. Deutsch. New York: Academic Press.
- Simon, H. (1969). The Architecture of Complexity. In The Sciences of the Artificial. Cambridge: MIT Press.
- Sloboda, J. (1985) The Musical Mind: The Cognitive Psychology of Music. Oxford: Clarendon Press.
- Smith, B. C. (1982) Reflection and Semantics in a Procedural Language. Ph.D. dissertation. Technical Report MIT/LCS/TR-272, Cambridge, Mass.: MIT.
- Smith, L. (1972) SCORE - A Musician's Approach to Computer Music. Journal of the Audio Engineering Society, 20.
- Todd, P.M. & D.G. Loy (Eds.) (1991) Music and Connectionism, Cambridge, Mass.: MIT Press.
- Wessel, D. (1979) Timbre space as a musical control structure. Computer Music Journal 3(2).
- Winograd, T. (1975) Frame Representations and the Declarative/Procedural Controversy. In Representation and Understanding: Studies in Cognitive Science, edited by D.G. Bobrow and A.M. Collins, New York: Academic Press.

NOTES

¹ Distributed representations (e.g. connectionist networks), in this sense, manipulate symbols of an unusual kind. An individual unit of such network does not implement an identifiable symbol; a meaningful representation only exist at a level made up of a number of units.

² Simon (1969) describes *nearly decomposable systems* as having the property "the short-run behaviour of each of the component subsystems is approximately independent of the short-run behaviour of the other components" (p. 100).

³ Allen's theory (1983), describes points as intervals that are durationless, i.e. a duration less than a value ϵ , adjusted to the reasoning task.

⁴ It has been shown that structure is essential in the performance of the continuous and discrete aspects of musical time (e.g. Clarke, 1987, 1988). Therefore a complete representation of time should facilitate the expression of these aspects in terms of structure to be of any perceptual or musical relevance (see Desain & Honing, 1991a).

⁵ A complete overview of all musical constructs will quite likely turn out to be a large, if not infinite collection, but they probably can be grouped into a considerably smaller set of proto-typical relations, with their specific characteristics being modeled as refinements of a particular structural type (see issue on Musical structure: association with time intervals and their constraints essential).

⁶ The constraints on the time intervals, as shown in Figure 1b and 2b, give a raw characterization of the example structures, just for comparison. For a more complete characterization of such structures the logic-based constraints of Allen (1983) are not enough. Other kinds of constraints are needed as well to be able to express relations like, for example, all bars have the same length, or, a bar is half the length of 'bars'.

⁷ These pages could be of different shapes and material, standing for structural descriptions of a completely different nature. This analogy was suggested by Morris Halle in a seminar at Sussex University in 1987 when talking about conceptual representations of linguistic structure.

⁸ Recent work done in the field of artificial intelligence on non-monotonic logic and truth-maintenance might therefore be applicable to music.

⁹ Since this article was written (autumn, 1990) work has been done on partial solutions of the issues presented above. Some of the issues on the representation of time have been resolved in a generalized concept of time functions (Desain & Honing, in press). A proposal for a specification and transformation formalism of expressive timing described in terms of structure is published as Desain & Honing (1991b).

TEMPO CURVES CONSIDERED HARMFUL

Peter Desain & Henkjan Honing

MARCH 1991
EDITED MAY 1991

Will be published as: Desain, P. & H. Honing. (1991). Tempo curves considered harmful. In "Music and Time", edited by J.D. Kramer. *Contemporary Music Review*. London: Harwood Press.

© copyright 1991, Peter Desain & Henkjan Honing

Center for Knowledge Technology
Lange Viestraat 2b
3511 BK Utrecht
The Netherlands

CONTENTS

Abstract..... 3

Keywords 3

In which we decided to have a good time, invited an expert, and had our first
disappointment..... 4

Tempo, Metre and Beat..... 6

Tempo, Timing and Structure..... 9

Wherein we looked at multiple performances, learned from a conductor and tried
different hierarchies but had no success. 9

Timing and Tempo, Patterns and Curves 11

Generative models 15

In which we investigated discrete patterns and continuous curves, tried
interpolation and failed again 15

Subjective Time, Duration and Tempo Magnitudes 16

Objective Time, Duration and Tempo Measurements..... 20

Epilogue 20

Acknowledgements..... 22

References..... 22

ABSTRACT

In the literature of musicology, computer music research and the psychology of music, timing or tempo measurements are mostly presented in the form of continuous curves. The notion of these tempo curves is dangerous, despite its widespread use, because it lulls its users into the false impression that a continuous concept of temporal flow has an independent existence, a musical or psychological reality, and that time can be perceived independent of events carrying it. But if one bases a transformation or manipulation of timing on the implied characteristics of such a notion, one is doomed to fail.

KEYWORDS

representation of time, tempo curves, expressive timing

TEMPO CURVES CONSIDERED HARMFUL

Peter Desain & Henkjan Honing

In which we decided to have a good time, invited an expert, and had our first disappointment.

Not so long ago we decided to spend a Christmas holiday studying music and its performance. One of us is an amateur mathematician (M) and the other one likes to delve into old psychology textbooks (P), and because we enjoy impressing each other with new facts and insights, we often find ourselves in vehement discussions. Therefore we thought we might have a pleasant and peaceful time by putting our beloved hobby horses aside and embark upon a subject about which neither of us knew much: the timing aspects of music. We became interested in this field because we had noticed, while playing with the computer, our favourite toy, that adding just a bit of random timing noise to a program that played a score in an otherwise metronomically perfect way, made the music much more pleasant to listen to. It seemed as if we could make more sense of it. But we suspected that there was more to timing and expressive performance than adding bits of noise, so we invited a mutual friend who is a retired professional pianist to spend Christmas in our small but well equipped laboratory. Our friend has a great love for the piano and its music, but is completely ignorant of the advances of modern technology. To demonstrate to him our latest sequencer program we asked him to play the theme from the six variations composed by Ludwig van Beethoven on the duet *Nel cor più non mi sento*, the score of which we had lying around (see Figure 1).

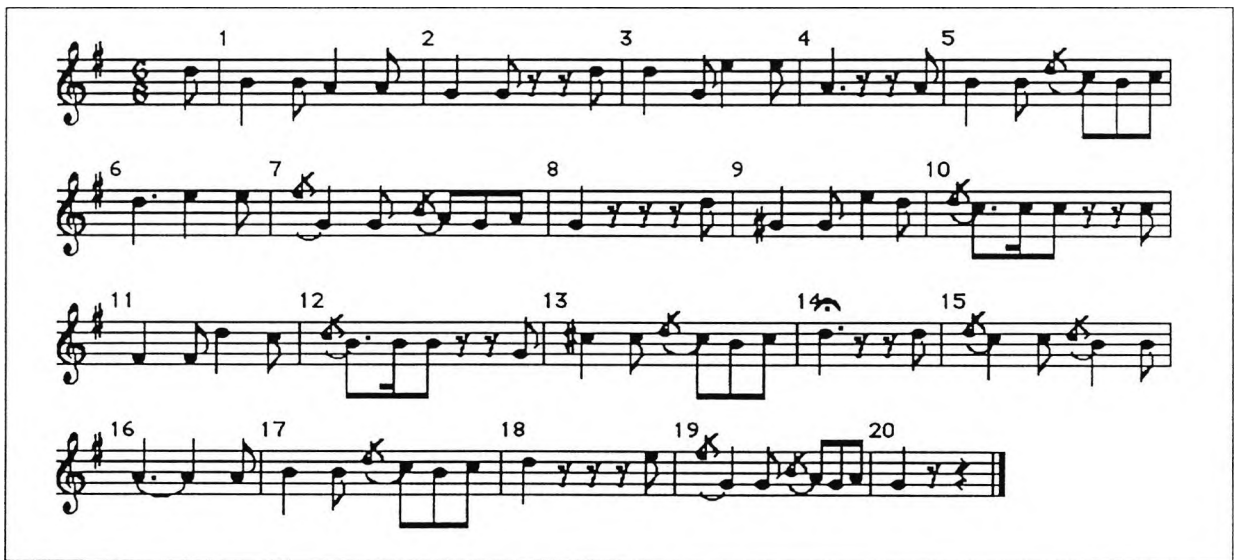


Figure 1. Score of the theme of *Nel cor più non mi sento*.

Even though he was somewhat disturbed by the touch and harpsichord-like sound of the electronic piano, he was quite fascinated with the possibility of recording and playing back on the same instrument. Enthusiastically we told him that this system was more than just a modern version of the pianola: 'You can examine and change every detail you want; for instance, inspect the timing, accurately to the millisecond, add and remove notes, make notes longer or shorter, or louder or softer, and so on and so forth.' Our friend became quite excited and asked: 'Could your machine play my performance in a minor key?' We were a bit put off by the simplicity of his demand, but patiently demonstrated the key-change feature. After hearing his performance with the key changed to G minor our friend was not impressed. 'O dear, I'm afraid this sounds much too hasty. For example, the "dramatic" e-flat in bar 3 needs more time. Let me play it in minor for you.' When we looked at the timing data of his new performance it indeed showed a different pattern. Upon noticing our disappointed faces our friend remarked 'this was not a minor change; it really turns it into another piece. We did not expect your device to know about that, did we?' We kept silent. 'But your machine can undoubtedly play the same piece at a faster tempo.' That set us in motion again. We changed the setting of the tempo knob to a tempo one-and-a-half times as high and pushed the play button. The face of our friend again did not show the expression we had hoped for. 'I'm awfully sorry, but this is not right! It sounds like a gramophone record played at the wrong speed, but without changing the pitches.' Suspiciously, we wanted some proof for his crude statement and asked him to play it the way he thought it ought to be performed. His version at the higher tempo was indeed different. We had to admit that it sounded more natural than our artificially speeded-up version. What made it sound so much better? We tried to unravel this mystery by examining the timing of the onsets and the offsets of the notes, since these were the variables that could be altered with our electronic keyboard, just like a real harpsichord.

Tempo, Metre and Beat

Temporal pattern is a series of time intervals, without any interpretation or structure.

Rhythm is a temporal pattern with durational and accentual relationships and possibly structural interpretations (Dowling & Harwood, 1986).

Beat refers to a perceived pulse marking off equal durational units (Dowling & Harwood, 1986, p. 185). They set the most basic level of metrical organisation. The interval between beats is sometimes called a "time-span" (Lerdahl & Jackendoff, 1983), or, less abstract, beat duration, beat period or metrical unit (Longuet-Higgins & Lisle, 1989).

Metre involves a ratio relationship between at least two time levels (Yeston, 1976). One is a referent time level, the beat period, and the other is a higher order period based on a fixed number of beat periods, the measure. It imposes an accent structure on beats, because beats initiating higher level boundaries are considered more important.

Tempo refers to the rate at which beats occur (often expressed as beats per minute), and is therefore closely linked to the metrical structure.

Density is used to refer to the average presentation rate taken across events of different duration (i.e. events per second) when a piece has events of different durations and the beat is hard to determine unambiguously, if at all (Dowling & Harwood, 1986).

It is important to note that rhythm, tempo, metre and density can be conceived independently: it is possible to maintain the same tempo while changing density; for example, a musical fragment can have a lot of embellishments (i.e. have a high density) and still be perceived as having a slow tempo. Furthermore, rhythm can exist without a regular metre and any type of rhythmical grouping can occur in any type of metrical structure (Cooper & Meyer, 1960).

Tactus is the tempo expressed at the level at which the units (beats) pass at a moderate rate (Lerdahl & Jackendoff, 1983). This rate is around the "preferred" or "spontaneous" tempo of about 100 beats per minute (Fraisse, 1982).

Our sequencer, a very recent version, had a separate tempo track. In this track, the tempo can be changed from fragment to fragment, even from note to note. With this feature we could put the original score on one track and the timing of the performance, expressed as tempo changes per note, on the tempo track, although it took quite a bit of calculating and editing by hand. After a while we had completely recreated the original performance, but now as a score plus a separate track of expressive timing

information. This tempo track looked like the graph in Figure 2a (for clarity we show only the timing of the melody). We could now compare the timing of this performance with the one played at tempo 90 (see Figure 2b). Their form was quite different even by visual inspection, although our ears were, of course, the only valid judges.

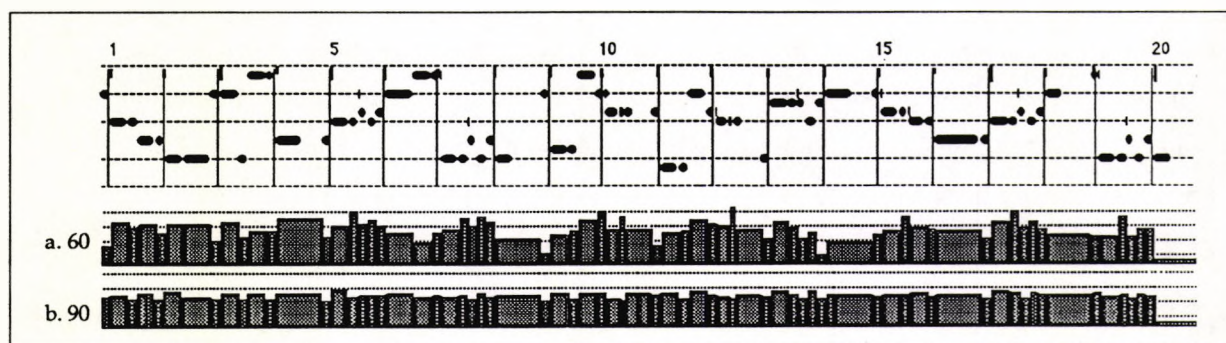


Figure 2. Tempo deviations in the performance of the theme at tempo 60 (a) and at tempo 90 (b).

What had happened? The sequencer had speeded everything up by the same amount (which we all agreed sounded awkward), while in the performance the expressive timing appears not to scale up everywhere by the same factor. Our friend adapted his rubato according to the tempo, which he explained to us as: 'My timing is very much linked to the musical structure and what I want to communicate of it in an artistic manner to the listener. If I play the piece at another tempo, other structural levels become more important; for instance, at a lower tempo the tactus will shift to a lower level, the subdivisions of the beat will get more "in focus", so to say, and my phrasing will have much more detail.' After some scratching with pen on paper, M found a quite elegant way of representing these changes using simple mathematics. We took the time interval between the onsets of every two succeeding notes and calculated the ratios of these time intervals in the two tempi. If the expressive timing pattern would scale-up linearly, we would find the ratios for all the notes to be around the ratio between the two tempi, and most ratios were indeed around 1.5. There was some variance around that factor, though, and we thought that could be explained by the more elaborate short-span phrasing at the lower tempo. But, even more noticeable was the fact that for some notes the ratio was close to 1. We found that these notes were notated as grace notes in the score. They did not change at all when performed at another tempo. We also found that not all grace notes behaved like this. For example, the two grace notes that cover an interval of a sixth, in bar 7 and 19, were timed like any other note: they were actually played in a metrical way. Our pianist got really excited about our observations. He pointed at grace notes in the score that were notated in the same way,

but that needed a different interpretation, and he started to lecture about the different kinds of ornaments, so popular in the eighteenth century, the difference between *acciaccatura* and *appoggiatura*, 'ornaments that "crush in" or "lean on" notes', about their possible harmonic or melodic function changing their performance, and so on and so forth. When he noticed that we were getting bored with his lengthy historical observations, he woke us up again with a new, sharp attack on our beautiful sequencer program: 'It might be forgivable that your program cannot play the onsets of ornaments correctly, but it also murders the articulation of most notes, especially the staccato ones. And have you heard what the program did to my detailed colouring of the timbre of chords?' Well, in fact, we had not, but we could well understand that the timbral aspect brought about by the chord spread (playing some notes in a chord a tiny bit earlier or later than others) was not kept intact when all timing information is just scaled by a certain factor. And we did not even dare to play the performance again at a lower tempo, afraid that each chord would turn into an arpeggio.

So our sequencer was not so wonderful after all. It could not be used to *change* something, not even such a minor thing as the key in which the piece was played. Again our pianist explained that a change of key was not a minor thing. The minimal variation that he could think of was the repetition of bars 5-8 at the end of the theme. 'The only difference between them is the fact that the second segment is a repetition of the first, and I even expressed that minimal aspect by timing. This problem is exacerbated if the difference between two sections is the overall tempo. Then detailed knowledge about structural levels, articulation, timing of ornamentations and chords, is indispensable.' We had to agree. How dumb of us, after all, to assume that a tempo knob on a commercial sequencer package could be used to adjust the tempo.

Tempo, Timing and Structure

In principle, timing can be linked to any musical structural concept. The most concrete of those are the following.

Although the most obvious *metrical units* are bar and beat, this strictly hierarchical structure may extend above and below these levels. Special expressive marking of the first beat in the bar, either by timing, dynamics or articulation, is a common phenomenon (Sloboda, 1983).

Phrases may not be ordered in a strict hierarchy, and may cut across metrical structure. Phrase final lengthening is the most well-known way in which they are treated (Todd, 1989)

A large proportion of the timing variance can be attributed to *rhythmical groups* (Drake & Palmer, 1990). Some standard rhythmical patterns, like triplets, seem to have a preferred timing profile (Vos & Handel, 1987).

Small timing asynchronies within a *chord* (called chord spread) are perceived as an overall timbral effect - the actual timing pattern is hard to perceive.

Ornaments, like *grace notes* and *trills*, can be divided in *acciaccatura*, so called timeless ornaments, and *appoggiatura*, ornaments that take time and can have a relatively important harmonic or melodic function. The former normally falls outside the metrical framework, the latter tends to get performed in a metrical way.

The independent timing of individual *voices* is sometimes hard to perceive because their components are immediately organised by the perceptual system in different streams (Bregman, 1990). This is not the case with (almost) simultaneous onsets that result in clear timbral differences. This can be heard in ensemble playing where often the leading voice takes a small lead of around 10 ms. (Rasch, 1979).

Any *associative relation*, e.g. between a musical fragment and its repetition, can be given intentional expression by using the same or different timing patterns.

Wherein we looked at multiple performances, learned from a conductor and tried different hierarchies but had no success.

But we were convinced we could make our friend happy, and proposed to program some additions to the sequencer ourselves. We showed him a video tape about research done at MIT by Barry Vercoe and his collaborators on computer accompaniment of a real musician. In this project the computer is given a score and several performances of the piece. With that information it can be "trained" to follow and accompany the musician.

Not that we were trying to do that, but we could use the idea to annotate each note in the score with its deviation in the performance, in our case in different tempi. Our friend friendly agreed to perform the Beethoven theme at four different tempi that were musically acceptable to him. We saw again that some notes exhibited a large change when tempo is changed, while others were less influenced by the tempo. But we could now use statistical methods to derive the right timing information for each tempo from this data. Our friend, who started to develop a little bit of suspicion, asked: 'Will that solve playing at different tempi then?' We were not quite sure. We definitely had more information now, but the representation of the music was still flat; no structural information was provided. It seemed we could not avoid incorporating some organisation above the note level into our program. Our friend agreed with a smile that was almost saying: 'are you stupid or am I?' We got a bit nervous. But after some discussion he agreed to concentrate on the timing of simple structural units like beats and bars only, leaving the note by note details aside for the moment.

Then we remembered Max Mathews working at CCRMA, Stanford University, who does important work in conductor systems (sort of the opposite of what Vercoe is doing). He made a system where one can conduct a sequencer on the beat level, which was just what we needed. The idea of a conductor shook our friend up; that sounded a much better approach than all those statistics we tried to explain to him before. We gave our friend an electronic baton, connected to our sequencer, and asked him to conduct the piece. In the score in the sequencer the beats were marked. The program followed the conductor by aligning each conducted beat with the corresponding mark in the score, and it tracked the tempo indicated by the conductor in doing so. At the high tempo, beating the baton very quickly, it seemed all right, but at the moderate tempo it was impossible to steer the timing deviations within the beat. 'It sounds too jumpy,' our friend complained. Since the beat level of the system of Mathews is arbitrary (he calls it 'generalised'), we annotated the score with marks at a lower metrical level, which alleviated the problem a bit. But, as our friend was still complaining about the controlability, we eventually ended up by marking each note in the score. This gave complete control at last, though our poor pianist, out of breath by the acrobatics needed to draw each note out of the sequencer by means of a single baton, made a cynical remark about the wonderful invention, which we may have heard of, called a keyboard. We became a bit vapid and proposed to help our conductor by connecting three MIDI batons to the computer, the first two used by us to time the bars and the beats, and the third to be used by our friend to fill in the details, using batons inter-connected with a complex mechanism of wires, to keep the timing at all levels consistent. We fantasized for some

time about a whole orchestra of conductors, leading one pianist before them. It was clearly time for a tea break.

Timing and Tempo, Patterns and Curves

In studying *timing deviations* a first distinction should be made between non-intended *motor noise* and intended *expressive timing* or *rubato*. The first category deviates in the range of 10 to 100 ms; the latter can deviate up to 50% of the notated metrical duration in the score.

Expressive timing is continuously variable and reproducible (Shaffer, Clarke & Todd, 1985) and clearly related to structure (Clarke, 1988; Palmer, 1989).

It is important to note that there is interaction between timing and the other *expressive parameters* (like articulation, dynamics, intonation and timbre). For example, a note might be accented by playing it louder, a fraction earlier than expected or by lengthening its sounding duration. Which method of accentuation is used is difficult to perceive, even when the accentuation itself is obvious.

To refer to expressive timing, in computer music the term *micro tempo* is often used, comparable to the term *local tempo* used in the psychology of music (the tempo changes from event to event, expressed as a ratio of a performance time interval and a score time interval). For clarity, the term *timing* would be more appropriate here. It specifies the timing deviation on a note-to-note basis and is often referred to as the *expressive timing profile* (Clarke, 1985; Shaffer, 1981; Sloboda, 1983), *timing pattern* or *rubato pattern* (Palmer, 1989).

In these patterns, points are often connected, either stepwise with straight line segments or with a smooth interpolation, yielding a *timing curve*. Only the first representation maintains a proper relation with the time map in which points are connected with line segments. These continuous time maps are used by Jaffe (1985) and most people of the computer music community. Time maps can be superimposed, using one for each voice.

Time maps can also be constructed for uniformly spaced units in the score like bars or beats. The corresponding duration patterns form a true *tempo pattern*. The points in these patterns can be connected by line segments, yielding so called *tempo curves*. Some authors insist on stepwise tempo changes, like Mathews (Boulanger, 1990), in which they are linked to one level of the metrical structure.

Over tea our friend told us about a series of programs on BBC radio, presented by the English conductor Denis Vaughan, on the composer's pulse he used in conducting. The pulse is a hierarchical, composer specific way of timing the beats. This pulse was an idea proposed and actually programmed by someone working in Australia. We went to our library and looked for some references that might tell us more on this composer's pulse. We ran into a collection of articles by Manfred Clynes who had invented the notion. This pulse, coincidentally, had precisely the characteristics we were looking for: hierarchical tempo patterns linked to the metrical structure. It basically entailed a system of automated hierarchical batons, and reduced the complexity further by postulating a fixed pattern for each baton. We took a final sip of our tea and hurried back to the lab and added Clynes' Beethoven 6/8 pulse as tempo changes in the tempo track to our sequencer. It divided the time for each bar into two unequal time intervals for the first and second half-bar and divided each half-bar into 3 unequal parts, one for each beat. With some adjustments here and there, we had our program running in no time. We called in our musical friend from the library to provide some professional judgements. He was definitely not unhappy with the result. 'This sounds much better than the things I've heard before,' he said.



Figure 3. Score of the first variation of *Nel cor più non mi sento*.

'Let's do the first variation, and see how our system performs it,' our friend said, far more optimistic now. He was talking about "our" system. This was a good sign. 'This variation is written in an ornamental style,' our friend explained, while we loaded the score of the first variation (Figure 3) into our system and created the tempo track containing the Beethoven pulse for this material. 'The metrical and harmonic structure is the same for both theme and the first variation. The only difference is that there are more "ornamental" notes added,' he said in a patronizing tone. When everything was set we played him the result. 'Well, this is disappointing,' was his short and decisive answer. After seconds of uncomfortable silence he added, 'it lacks the general phrasing and detailed subtlety I think is essential to make it an acceptable performance. The rhythmical materials of the theme and the first variation are different. The sixteenth notes of the variation ask for a different kind of timing than the mainly short-long, short-long, short-long rhythm of the theme. This pulse plays only with the metrical structure, but musical structure has far more to offer than that.' So the composer's pulse could not just be mapped onto any rhythmic material. Furthermore, it only linked timing to the meter, and, as our friend made clear, phrasing and other musical structure was ignored.

That rang a bell. We remembered one of the articles by Neil Todd on a model of rubato, linked to phrase structure. His proposal is very similar to Clynes; it explains timing in terms of a hierarchical structure, but now phrase structure is the basic ingredient. The beat is again the lowest level; below that no timing is modelled. The abundance of mathematical notation in Todd's articles did not put off our amateur mathematician. Quite the contrary. 'This, on first sight, will give us a solid basis to work with. What he states here is that, if you remove all the constants from the formula, it is actually quite simple,' M said. 'Todd proposes to attach a parabola to each level of the hierarchical phrase structure, and sum their values to calculate the beat length.' He simplified a formula, found an error on the way and finally the model became easy to implement. We were quite conscious of the fact that we were the first really to hear Todd's model (he himself had never listened to it). It did not sound very pleasing because this model was expressed in terms of the phrase structure only (based on the idea of systematically lengthening the end of a phrase in a hierarchical way), and because it lacked all expressive timing below the level of beats.

Longing to show our collaborator that the computer could, in principle, also calculate detailed note-by-note timing, we looked for a model that would provide these. Happily we found masses of rules for those subtle nuances in the articles of Johan Sundberg and his colleagues. These rules formulated simple actions, like inserting a

small pause in between two notes or shortening a note. The actions had to be performed if the notes matched a certain pattern, such as constituting a pitch leap or forming part of a run of notes of equal duration. In fact there were so many rule sets proposed in his articles that we got a bit lost in the details, but it has to be said that some rule-cocktails really seemed to work for our piece. Especially if their influence was adjusted to effect a subtle change only, the music gained some liveliness. But because these rules are based on the surface structure of the music only we could predict the judgement of our musical expert by now. And indeed he did not even bother to comment on the artificially produced performances. Instead he kindly reminded us that we might give up looking for a system that enabled us to generate a "musically acceptable" performance, given a score (that is what Clynes, Todd and Sundberg are aiming at), for the simple reason that we already had an "acceptable" performance, namely his own. It was true, the initial aim of our endeavour was to find ways of manipulating the timing in a musically and perceptually plausible way, given a score *and* a performance. Because the simple representations we had used proved unsuccessful, we had been sidetracked by studying even simpler representations that could at most model a small aspect of our friend's performances. We decided to close the session, look for more details in the literature, and give it another try the next day.

Generative models

Clynes (1983; 1987) proposes composer specific and metre specific, discrete tempo patterns. This so called composer's pulse is assumed to communicate the individual composer's personality. E.g. in the Beethoven 6/8 pulse the subsequent half-bars span 49 and 51% of the bar duration and each half bar is divided again in 35, 29 and 36%. Clynes is opposed to analysis of performance data: the pulses stem from his intuition. Repp (1990) has undertaken a careful evaluation of this model.

Todd (1985; 1989) proposes an additive model in which beat duration is calculated as a summation of parabola shaped curves, one for each level of hierarchical phrase structure. He complemented the model with an analysis method that calculates phrase structure from beat durations.

Sundberg et al. (1983; 1989) proposes a rule system to generate expression from a score based on surface structure. His research was done in an analysis-by-synthesis paradigm and captures expert intuition in the form of a large set of these rules. An example of a rule is "faster uphill": A duration of a note is shortened if it is preceded by a lower pitched note and followed by a higher pitched one. Van Oosten (1990) has undertaken a critical evaluation of this system.

In which we investigated discrete patterns and continuous curves, tried interpolation and failed again.

We found all kinds of references in the literature and read a lot that evening. It was amazing to find how much work actually was done on a problem that we had thought was not a problem at all. We became a little bit more conscious of the whole thing. It looked as if P's hobby horse, psychology, had to be given a chance. He explained that the perception of time had been modelled postulating a certain (often exponential) relation between objective time and experienced time. But this research had all been done with impoverished stimulus material, often consisting of just one time interval marked-off with two clicks. 'Other research,' P added, 'found that duration judgment depends on the way the interval is filled with more or fewer events, so unfortunately these simple laws cannot be directly applied to more complex material like real music.' Even P was disappointed with the results of his beautiful science. 'But psychology has something to offer to us here', he spoke in a defensive tone. 'Take a look at all the articles that present timing or tempo measurements in the form of continuous curves instead of just a scattergram of measurements. These curves more or less imply an independent existence, apart from the rhythmic material where they were measured

from. But psychological research has shown that one cannot perceive timing without events carrying it.' He found this convincingly argued in an article by the psychologist James J. Gibson called "Events are perceivable but time is not". 'Can you imagine perceiving a rubato without any notes carrying it?' P asked. 'And vice versa: "filling up" time by adding an event between two measured points is problematic, isn't it?' There seemed to be no possible argument.

Subjective Time, Duration and Tempo Magnitudes

Most psychophysical scales for time intervals are described by Stevens' Law, that relates the physical magnitude of a stimulus to its perceived magnitude as perceptual-time = a-constant.physical-time^{b-constant}. The b value differs from one dimension to the other. For time duration b is commonly found to be 1.1, slightly over estimation of the interval. However, for intervals shorter than 500 ms it is found that b is around 0.5, the square root of its physical duration (Michon, 1975).

Humans seem to have a relatively poor ability for time discrimination of intervals presented without context. The just notable differences (JND) are in the range of 5-10% (Woodrow, 1951) with an optimum near 600 ms intervals. However, in the context of a steady beat, the JND's are around 3% with the same optimum interval (Povel, 1981).

Much research was done on the existence of a spontaneous tempo, preferred rate or natural pace (Fraisse, 1982). This tempo should occur as a preferred rate of spontaneous tapping, and material presented at that rate should be easy to perceive and remember. There is weak, but converging evidence for the existence of such a rate, again with intervals around 600 ms. There is no consistent evidence for physiological correlates like heart rate.

There has been quite some research done on the influence of different dimensions on time perception, mainly in the fifties. Evidence was found that, in general, the higher pitched the sound the longer the percept (Cohen et al., 1954), and the same holds for louder sounds (Hirsch et al, 1956). Evenly divided intervals seem longer than irregular divided ones (Ornstein 1969).

Time intervals shorter than 120 ms, preceded by a physically shorter neighbour time interval, are underestimated to such a remarkable degree that one can speak of an auditory illusion (Nakajima et al., 1989).

We decided to do the acid test using a feature of the sequencer program. In this program it was possible to copy tempo tracks from one piece to the other. We applied

the tempo track of the original performance of the theme (see Figure 2) to the score of the first variation. The result was poor; even we could hear that. The timing made sudden jumps, like a beginner sight-reading and hesitating at unexpected points because of a difficult note. The expressive timing pattern found in the theme did not “fit” the variation. Our friend’s performance of the variation was much smoother and had gestures on a larger scale, as far as we were able to judge (Figure 4). Also, the other way around, taking the timing data from the variation and applying it to the score of the theme had the same awkward effect. It seemed impossible to just add or remove notes using these stepwise tempo curves. We felt stupid again for having assumed that the independence of tempo tracks in the sequencer made musical sense. But it made us look in the literature for alternatives.

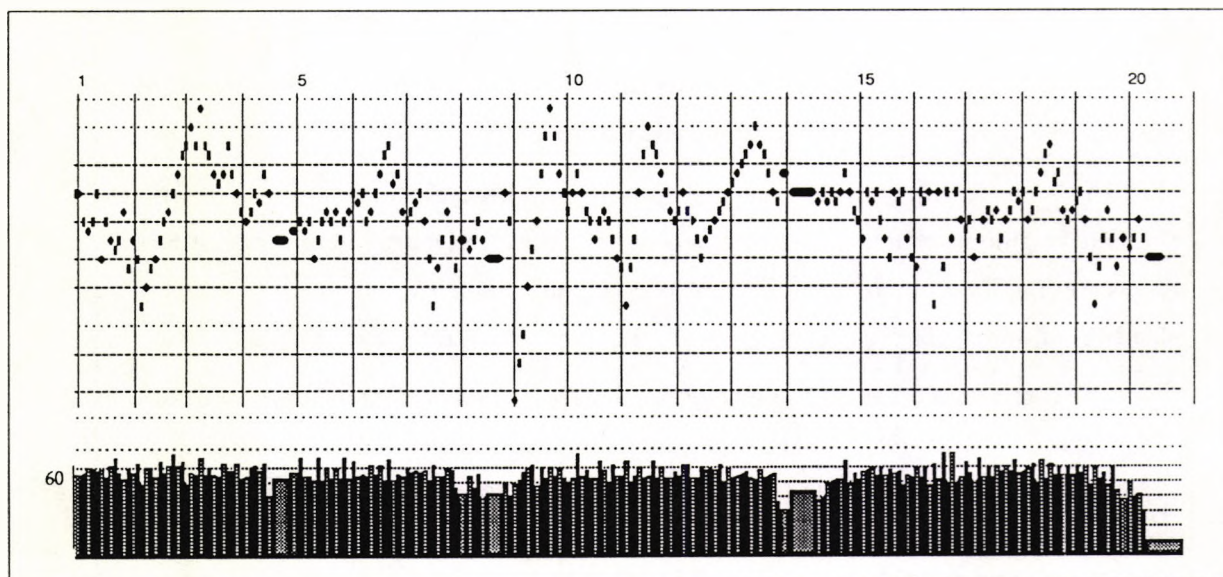


Figure 4. Tempo deviations in the performance of the variation at tempo 60.

The answer was not far away. In the field of computer music research continuous rubato curves were used almost by default. We decided to take the path of the continuous timing functions, hoping it would get rid of this awkward “jumpiness”. Thus M’s hobby horse was brought out again. ‘Functions are far easier to handle. One can calculate, given the right kind of function, a good timing curve for every piece,’ M argued convincingly. This combined approach of formality (in the mathematical sense) and pragmatics reminded us of a method developed by David Jaffe of CCRMA to model the timing of different parts of a computer orchestra. Jaffe wanted the different instruments to have their own timing, but they had to synchronise at specific points as well. By using a time map, instead of tempo changes, coordination and synchronization

became possible. 'What he actually does is to specify the timing for each event by means of a function from score time to performance time,' M explained, 'a blatantly simple idea indeed: to integrate velocity or one-over-tempo, as Jaffe calls it, to get time. This of course restrains the possible functions one can use to make up such a time map; they have to increase monotonously and one must be able to calculate a first derivative.' This was again a method, among many others, in which different authors presented their ideas of tempo curves (see Figure 5). We tried to bring some order to the ways the different representations were used.

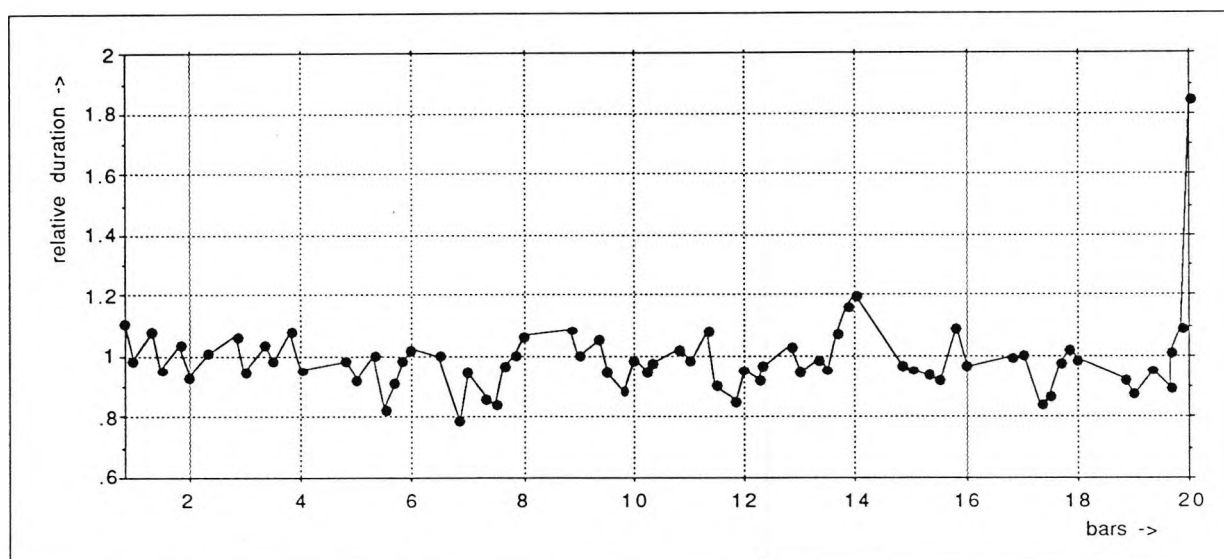


Figure 5. A typical so called "Tempo Curve" with duration factors for each note as a function of metrical time.

Soon M gave up, stating that it was a hopeless mess; no two authors used the same dependent and independent variables and measurement scales. And while in the end all the information needed could be extracted from most presentations, it was a difficult job, the more so because of the confusion in terminology. We decided to return to the practical application of the time map. We adapted the sequencer's tempo track to contain a time map (composed of line segments) instead of the discrete tempo changes we had used before. We then applied this continuous curve to the variation and had our pianist judge it. He thought it was much better than the direct application of the discrete curve of the theme to the variation. The interpolation (with line segments) did improve the smoothness of the timing, but he still complained about the sudden tempo jumps at the junctions of the curve. M remarked that one could restrict the allowed tempo map functions further or smooth the existing function, for instance, with splines. This brought us to an article describing work done at IRCAM by David Wessel and others, which indeed proposes the use of splines. We took an algorithm we had lying

around that did splines and added it to our tempo track algorithm. And there it was: with some twiddling of the parameters we could interpolate the timing pattern of the theme for its use on the variation. We almost thought that with this interpolation we had proven Gibson wrong. There was a smooth sense of timing in between events, and if one is smart enough one can tap it and hook new events onto it in a reasonable way. But our musical friend did not agree 'Reasonable?' he reacted angry, 'it sounds reasonable, yes, but your numerical calculations have nothing to do with the way I played it, whatsoever. The musical structure, my dear friends, remember the musical structure. How often do I have to repeat this. Timing *is* related to structure!' We suggested to him a cup of tea, in the hope that this would calm him down.

Objective Time, Duration and Tempo Measurements

When an event happens (an onset of a note) one can measure the *real time* elapsed since the beginning of the piece (called *performance time*) and also the point in the score where this onset was notated (called *score time*). The latter can be measured either in seconds (taking the tempo marking in the score serious, or normalising the total score length to the performance), in metrical units like beats or quarter notes (called *metrical time*), or as an event count (called *event time*). The last loses so much information that the timing pattern cannot be reconstructed without reference to the score.

Performance time can be shown as a function of score time (called a *time map*), or vice versa. In these representations it is easy to spot (a)synchronies between voices because they depict points in absolute time.

Calculating differences between subsequent performance times in a time map makes the step from time to duration. Because in such a representation it is difficult to compare notes of different nominal duration, a proportional measure is better. It makes the step from duration to relative duration by dividing two corresponding durations. In case a performance duration is divided by a score duration, this forms a series of duration factors (often misleadingly called tempo). This measure is mostly notated in a graph with the independent axis labelled with metrical or event time. In the case of the inverse calculation, the ratios form the velocity, the local speed of reading the score.

In both cases the measured points are often filled in with line segments - implying the existence of a tempo measurement in between events. This is misleading - the more so because integration does not yield the original time map again.

Gabrielsson (1974) uses note duration expressed in proportion to the length of the bar. This allows for comparison with exact note values in different meters. The method might be generalizable to study timing at different levels of structure.

Tempo is sometimes presented on a logarithmic scale; this is a first step towards the use of subjective magnitudes.

An interesting hypothesis was given by Brown (1979). He argues that a musician makes use of a collection of discrete tempi: a collection of discrete physically possible tempi, where the choice is defined by musical and performing factors.

EPILOGUE

What this partly fictitious story (the characters are fictitious, but the examples and arguments are real!) shows is that we have to be aware of the Tempo Curve. Of course

one should be encouraged to measure tempo curves and use them for the study of expressive timing. But it is a dangerous notion, despite its widespread use and comfortable description, because it lulls its users into the false impression that it has a musical and psychological reality. There is no abstract tempo curve in the music nor is there a mental tempo curve in the head of a performer or listener. And any transformation or manipulation based on the implied characteristics of such a notion is doomed to fail.

That does not mean that generic models that represent timing in terms of some sort of structure, even when they describe just a fraction of the many aspects of expressive timing, do not constitute a valuable contribution to the field. They only have to be seen in a proper perspective in which their limitations are understood as well. It also does not mean that certain features in computer music software and commercial sequencers should be forbidden. Their mere existence at least makes the realisation of their limited worth evident.

It should be noted here that the views expressed in this article comply more or less with the British school of expressive timing research (E.F. Clarke, H.C. Longuet-Higgins, L. Shaffer, J. Sloboda and N. Todd), in which the link between structure and timing is paramount. There are alternative views developing at the moment, denying such a strong link (Kendall & Carterette, 1991). We hope this controversy will eventually lead to more understanding of this wonderfully complex aspect of music performance.

In reality the experiments were done using POCO, an environment for analysing, manipulating and generating musical expression (Honing, 1990), which took a bit longer to build than one Christmas.

The holiday was almost over now and we felt that we had not found out many useful things. Our musical friend announced that he would go back to his own piano. He thanked us for the interesting sessions, from which he had learned a lot. But underneath these friendly remarks we could hear the cynicism. He advised us in a fatherly way to get rid of our research papers and start reading biographies of famous composers, in which the true facts about music and its performance could be found. This made the feeling of disappointment even more pronounced. But in a last irrational attack of bravery, we decided not to give in yet and we invited him to come back next Christmas, and to bring his biographies if he wished.

To be continued ...

ACKNOWLEDGEMENTS

Thanks to Boy Honing and Mariken Zandvliet for their performances of Beethoven. Thanks to Bruno Repp for information on Clynes' model and to Shaun Stevens for his help with the English language. We would like to thank also all the researchers mentioned, for their contribution to the field of timing in music. We are very grateful to Eric Clarke who made it possible for us to work for two years on research in expressive timing which allowed us to gain an insight into timing through our numerous discussions, and the British ESRC for their financial support throughout these two years (grant A413254004).

REFERENCES

- Boulanger, R. (1990) Conducting the MIDI Orchestra, Part 1: Interviews with Max Mathews, Barry Vercoe and Roger Dannenberg. Computer Music Journal 14(2).
- Bregman, A. (1990) Auditory Scene Analysis: the Perceptual Organisation of Sound. Cambridge, Mass: Bradford Books.
- Brown, P. (1979) An enquiry into the origins and nature of tempo behaviour. Psychology of Music 7(1).
- Clarke, E.F. (1985) Some Aspects of Rhythm and Expression in Performances of Erik Satie's "Gnossienne No.5". Music Perception, 2(3).
- Clarke, E.F. (1987) Levels of structure in the organisation of musical time. In "Music and psychology: a mutual regard", edited by S. McAdams. Contemporary Music Review, 2(1).
- Clarke, E.F. (1988) Generative principles in music performance. In Generative processes in music. The psychology of performance, improvisation and composition, edited by J. A. Sloboda. Oxford: Science Publications.
- Clynes, M. (1983) Expressive Microstructure in Music, linked to Living Qualities. In: Studies of Music Performance, edited by J. Sundberg. Stockholm: Royal Swedish Academy of Music, No. 39.
- Clynes, M. (1987) What can a musician learn about music performance from newly discovered microstructure principles (PM and PAS)? In Action and Perception in Rhythm and Music, edited by A. Gabrielsson. Royal Swedish Academy of Music, No. 55.
- Cohen, J., C.E.M. Hansell, & J.D. Sylvester. (1954) Interdependence of temporal and auditory judgements. Nature, 174.

- Cooper, G. & Meyer, L. B. (1960) The rhythmic structure of music. Chicago: University of Chicago Press.
- Dowling, W. J. & D. L. Harwood (1986) Music Cognition. London: Academic Press.
- Drake, C. and C. Palmer (1990) Accent Structures in Music Performance. manuscript.
- Fraisse, P. (1982) Rhythm and Tempo. In The Psychology of Music, edited by D. Deutsch. New York: Academic Press.
- Gabrielsson, A. (1974) Performance of rhythm patterns. Scandinavian Journal of Psychology, 15.
- Gibson, J. J. (1975) Events are perceivable but time is not. In The Study of Time, 2, edited by J.T. Fraser & N. Lawrence. Berlin: Springer Verlag.
- Hirsch, I.J., R.C. Bilger & B.H. Deathrage (1956) The effect of auditory and visual background on apparent duration. American Journal of Psychology, 69.
- Honing, H. (1990) POCO, An Environment for Analysing, Modifying and Generating Expression in Music. In Proceedings of the 1990 International Computer Music Association. San Francisco: CMA.
- Jaffe, D. (1985) Ensemble timing in Computer Music. Computer Music Journal, 9(4).
- Kendall, R.A. and E.C. Carterette (1990) The Communication of Musical Expression. Music Perception, 8(2).
- Lerdahl, F. & R. Jackendoff (1983) A Generative Theory of Tonal Music. Cambridge, Mass.: MIT Press.
- Longuet-Higgins, H.C. & E. Lisle. (1989) Modelling musical cognition. In "Music, Mind and Structure", edited by E. Clarke and S. Emmerson. Contemporary Music Review 3(1).
- Michon, J.A. (1975) Time experience and memory processes. The Study of Time, 2, edited by J.T. Fraser & N. Lawrence. Berlin: Springer Verlag.
- Nakajima, Y., T. Sasaki, R.G.H. van der Wilk, & G. ten Hoopen. (1989) A new illusion in time perception. Proceedings of the First International Conference on Music Perception and Cognition. Kyoto, Japan: The Japanese Society of Music Perception and Cognition.
- Oosten, P. van (1990) A Critical Study of Sundbergs' Rules for Expression in the Performance of Melodies. Proceedings of the Music and the Cognitive Sciences Conference, Cambridge.
- Ornstein, R.E. (1969) On the Experience of time. London: Pinguin.
- Palmer, C. (1989) Mapping Musical thought to musical performance. Journal of Experimental Psychology, 15(12).

- Povel D.J. (1981) Internal Representation of Simple Temporal Patterns. Journal of Experimental Psychology: Human Perception and Production. 7(1).
- Rasch, R. A. (1979) Synchronization in Performed Ensemble Music. Acustica 43(2).
- Repp, B. (1990) Further Perceptual Evaluations of Pulse Microstructure in Computer Performances of Classical Piano Music. Music Perception. 8(1).
- Shaffer, L.H. (1981) Performances of Chopin, Bach and Bartok: Studies in motor programming. Cognitive Psychology, 35A.
- Shaffer, L.H., E.F. Clarke, & N.P. Todd (1985) Metre and rhythm in piano playing. Cognition, 20.
- Sloboda, J. (1983) The communication of musical metre in piano performance. Quarterly Journal of Experimental Psychology, 35.
- Sundberg, J., A. Askenfelt & L. Frydén (1983) Musical Performance: A synthesis-by -rule Approach. Computer Music Journal, 7(1)
- Sundberg, J., A. Friberg & L. Frydén (1989) Rules for Automated Performance of Ensemble Music. Contemporary Music Review, 3.
- Todd, N. (1989) A Computational Model of Rubato. In "Music, Mind and Structure", edited by E. Clarke and S. Emmerson. Contemporary Music Review 3(1).
- Todd, N.P. (1985) A model of expressive timing in tonal music. Music Perception, 3.
- Vos. P. & Handel, S., (1987) Playing Triplets: Facts and Preferences. In: Action and Perception in Rhythm and Music, Edited by A. Gabrielsson Royal Swedish Academy of Music. No. 55.
- Woodrow, H. (1951) Time Perception. In Handbook of Experimental Psychology, edited by S.S. Stevens. New York: Wiley.
- Yeston, M (1976). The stratification of musical rhythm. New Haven CT: Yale University Press.

III

MICROWORLDS

TIME FUNCTIONS FUNCTION BEST
AS FUNCTIONS OF MULTIPLE TIMES

Peter Desain & Henkjan Honing

MARCH 1991
EDITED MAY 1991

Will be published as: Desain, P. & H. Honing. Time Functions Function Best as Functions of Multiple Times. *Computer Music Journal*. Cambridge, Mass.: MIT Press.

© copyright 1991, Peter Desain & Henkjan Honing

Center for Knowledge Technology
Lange Viestraat 2b
3511 BK Utrecht
The Netherlands

CONTENTS

Introduction.....	3
Framework of discrete musical objects.....	5
Continuous control.....	7
The problem	7
A solution	8
Control over compound objects	13
Time function composition	15
Specification by means of transformation.....	18
Microworld	19
Extensions	20
Articulation	20
Real-time control.....	20
Rubato functions and time maps.....	21
Advantages of "Time functions of multiple times"	21
Acknowledgements.....	21
References.....	22
Appendix.....	24

TIME FUNCTIONS FUNCTION BEST AS FUNCTIONS OF MULTIPLE TIMES

Peter Desain & Henkjan Honing

Center for Knowledge Technology
Utrecht School of the Arts
Lange Viestraat 2B
NL-3511 BK Utrecht

COCO Foundation
P.O.Box 1037
NL-3500 BA Utrecht

This paper presents an elegant way of representing control functions at an abstract level. It introduces control functions that have multiple times as arguments. In this way the generalized concept of a time function can support absolute and relative kinds of time behavior. Furthermore the possibilities of composition and transformation of time functions themselves is retained. The proposed solution has three main advantages. Firstly, for the human user the language is transparent, and no unforeseen interactions or side effects take place. Secondly, it is independent of host language and composition system and can be used in a variety of known environments (even in real-time systems). Finally, the method is easy to adapt to run on parallel architectures: each note can be handled by a different processor, without the need for information passing between them.

INTRODUCTION

In the early history of computer music composition (Loy, 1988) the systems available took either a monolithically continuous, signal processing inspired approach (Mathews & Moore, 1970; Berg, 1979), or used a discrete, note or event based technique (Hiller, Leal & Baker, 1966; Koenig, 1970). Although some early work stressed the importance of hybrid systems (Mathews, 1969; Buxton, Sniderman, Reeves, Patel & Baecker, 1978), this division became even more obvious once the rich domain of hardware and software, that became available for MIDI, had lured designers into building composition systems close to this note-based protocol. While MIDI allows for some rudimentary continuous control (i.e.

polyphonic aftertouch), most parameter changes affect either all sounding notes or all notes on a specific channel. With the recent advent of cheap signal-processing hardware that allows a more natural continuous control over parameters during each note's evolution, the quest for elegant constructs for composition languages supporting both worlds is on again. During this whole history some researchers foresaw these developments and made attempts to bridge the gap between both worlds, by stating the problems (Dannenberg, Dyer, Garnett, Pope & Roads, 1989; Huron, 1990; Honing, 1991) and proposing solutions to them (Dannenberg, McAvinney & Rubine, 1986; Anderson & Kuivila, 1989).

The main problem that arises is how *continuous* control functions should behave under specification and transformation of the *discrete* structure. A notorious example is the vibrato problem: a vibrato should not slow down if the note itself is elongated, but some extra vibrato cycles should be added to the pitch envelope. A discrete analogy of the vibrato problem is the drum roll, which should be extended by adding more hits, but its rate should not slow down. However, a glissando, specified by the same means of a continuous pitch envelope, should be stretched along with the note duration. A third example is an ornament (e.g. a mordant) that is invariant under transformation of the duration of the note.

Several solutions for these problems have been stated, but all are unsatisfactory. We will name a few. In Canon (Dannenberg, 1989) a collection of transformations on a fixed set of attributes is used, together with a way of communicating environment information to new transformations. With these constructs he explicitly solves the "drum roll problem", though in a non-trivial, almost procedural way. Dannenberg (1986; 1989) proposes the term "behavioral abstraction" for the ability to express these complex parametrized behaviors. Anderson & Kuivila (1989) describe a solution based only on global time, prohibiting the composer to think in terms of local constructs (most real-time composition systems have, besides actual time, no sense of time at all; see Desain & Honing (in preparation)). Solutions proposed for object-oriented composition systems (e.g. Pope 1987; 1989) suffer from a declarative/procedural confusion whereby transformations and musical objects form no orthogonal sets (each new transformation added has to take all object types *and* all existing transformations into account). This inevitably leads to the situation whereby some transformations cannot be done twice or some combinations cannot be done in an arbitrary order.

In this paper we will present an elegant and -once understood- obvious way of representing control functions at a more abstract level that simply evades all these

problems and yields a transparent specification of (discrete) musical structures with continuous control over their parameters. To be able to illustrate this approach, a simple framework language for discrete musical objects and their ordering in time is given, expressed in Common Lisp (Steele, 1990).

FRAMEWORK OF DISCRETE MUSICAL OBJECTS

Let us start by assuming a basic note object, and a basic rest object (called *pause*, to avoid clashing with the Common Lisp primitive *rest*) with some parameters specified by keywords:

```
(note :duration 2 :pitch 60 :amplitude 0.8)

(pause :duration 1)
```

The syntax is taken directly from Lisp, i.e. prefix notation with the function name before the arguments, the whole enclosed in brackets. Each argument is specified as a pair of a keyword and a value. The actual parameters allowed are irrelevant for the present introduction; a MIDI-based composition system might need other parameters than a signal-processing approach. Furthermore, the discussion on the magnitude scales for these parameters is ignored here. A simple assumption of a duration scale in seconds, a pitch scale in MIDI key numbers (with fractional part), and a $[0,1]$ scale for amplitude is assumed in the examples. Even the semantics of such expressions; whether they initiate processes, deliver data-structures that represent musical objects (i.e. event-lists), or are the actual procedures that output the material directly, is immaterial here. In the Appendix one possible implementation is given. We assume that not-mentioned parameters are defaulted to reasonable values (duration 1 second, pitch 60, and amplitude 1) for ease of use in the examples.

It must be possible to specify the timing of the basic musical objects, either by passing parameters for start time and duration directly (as is used here for the duration parameter) by means of parameters that place or move objects in time (Abbott, 1981; Dannenberg et al, 1986; Balaban, 1989), or by constructor functions that build or play musical objects in a distinct time order (Smoliar, 1980; Dannenberg, 1989). For the sake of simplicity, we will use the last approach with the *Sequential* and *Parallel* constructs that we used in the LOCO composition language (Desain & Honing, 1988) and which were elaborated as a basis for transformations in (Desain, 1990). These constructs mirror the sequential and parallel execution primitives originating from parallel language design. *S* stands

for a sequential ordering of its components, the whole structure ending after the last one, and *P* stands for a parallel structuring ending at the end time of its longest component. As an example, consider the following musical object. Its graphical piano-roll like representation is shown in Figure 1, where time is represented on the x-axis, pitch is represented on the y axis, and the amplitude of a note is represented by its shading.

```
(s (p (note :duration 2 :pitch 62 :amplitude 1.0)
      (note :duration 4 :pitch 65 :amplitude 0.7))
  (note :duration 5 :pitch 58 :amplitude 0.3))
```

Figure 1a. Example of a time structured musical object.

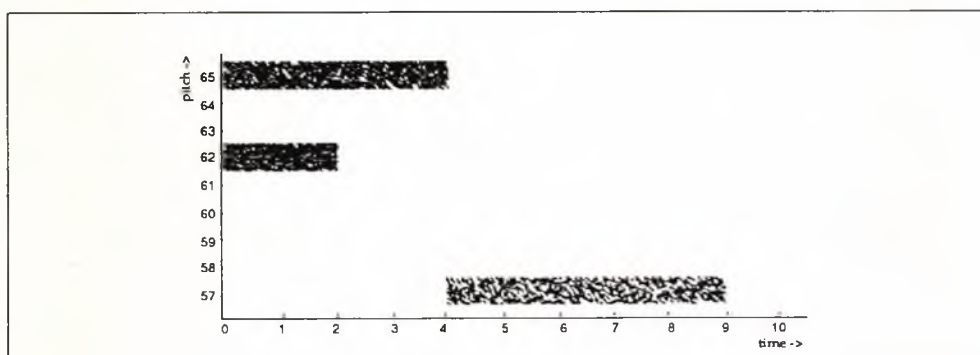


Figure 1b. Graphical notation of the musical object given in Figure 1a.

An extension of this approach allows for high level timing control for defining non-standard musical objects like grace notes. As these constructs are not essential for the present argument we refer to Desain and Honing (1988) for more details. For naming complex musical objects, forming parametrized families of objects and defining musical transformations we will use the standard procedural abstraction (function definition) facilities of the host language. An example of a compound musical object built by those means is shown in Figure 2.

```
(defun major-chord (duration key)
  (p (note :duration duration :pitch key)
     (note :duration duration :pitch (+ key 4))
     (note :duration duration :pitch (+ key 7))))

(s (major-chord 2 57) (major-chord 5 58) (major-chord 2 57))
```

Figure 2a. Use of procedural abstraction in defining a compound musical object.

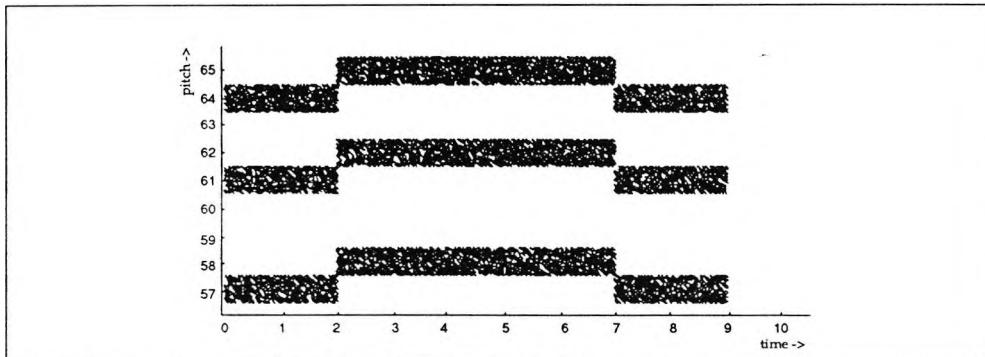


Figure 2b. Graphical notation of the musical object given in Figure 2a.

We introduce one basic transformation on the timing of fully constructed musical objects, that we can use to demonstrate the behavior of the time functions when the duration of the object they are linked to, is changed. The *stretch* transformation multiplies each duration of the enclosed objects by a factor.

```
(stretch (note :duration 1 :pitch 60 :amplitude 1) 2)
=
(note :duration 2 :pitch 60 :amplitude 1)
```

These preliminaries constitute a world rich enough to introduce continuous time functions and their problems, but it has to be kept in mind that the same solution can be used in most present composition systems, however different their notion of musical objects and collections thereof, and in whatever way the time relations between them are specified.

CONTINUOUS CONTROL

The problem

A most natural thought, when bored with note-based discrete systems, is passing to each note a continuously variable function of time as parameter for say pitch or amplitude instead of a constant value. The functions passed are functions of the actual time, and elegant ways can be given to build and transform them. Often

though, these functions have been regarded as control signals (resembling audio signals) even up to the point where a list of data points, interpreted at a fixed (low) sample rate has been called a function (e.g. Schottstaedt, 1983; Puckette, 1988), obscuring the highly abstract powerful possibilities of functions in their mathematical sense (exceptions are Rodet & Cointe, 1984; Dannenberg, McAvinney & Rubine, 1986; Anderson & Kuivila, 1989). But even when the full power of function specification is used, time functions are considered to be functions of actual time. This complicates the coupling of these functions to discrete objects and gives rise to the problem described before: the notorious vibrato problem and its discrete counterpart, the drum roll problem.

A related concern is the use of a relative or absolute start-point for the time base used. The use of an absolute time scale is sometimes preferred by composers because of the (false) impression of total control. However, it implies an envelope to be redefined each time it is used at another absolute point in time. This can simply be avoided by using a time base relative to the object under construction. This, of course, does not mean that the notion of absolute time control can be ignored. It is indeed indispensable when time relations with events outside the musical piece (say the midnight church bells) are to be taken into account, or, as is the case more often, when relations between different and independent musical objects have to be maintained (e.g. a synchronized vibrato between different voices).

A solution

The solution we propose is to make each control function a function of more parameters - each parameter reflecting a different aspect of time. As an example, we will develop this notion for time functions of three parameters: the absolute *start* time of the discrete musical object it is controlling; the absolute time *duration* of this musical object; and a relative *progress*, expressing in how far time has elapsed since the start time, relative to the duration (a number in the range [0,1]). Other choices are of course possible here (e.g. start time, end time and actual time). All these parameters will be passed their appropriate values automatically by the interpretative system. With this definition of a time function, the user can now choose to use some time parameters and ignore others to make time functions that behave differently when used for musical objects with different duration or start time.

As a first example, let us define an elastic ramp control, independent of an absolute start time, taking the full duration of the musical object to reach its final

value. We will use it to control the pitch of some notes, creating glissandi. The function `ramp` produces a linear time function of the three time parameters mentioned before. It ignores the absolute start time and duration parameters, depending only on the progress of the evolving note (a number between 0 and 1) to calculate its value. Figure 3 shows how the same ramp construct is used for notes of different duration -yielding an appropriate glissando- *and* how the musical idea is kept intact under a stretch transformation.

```
(defun ramp (from to)
  #'(lambda (start duration progress)
    (+ from (* progress (- to from)))))

(defun glissando-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (ramp 64 61))
      (s (pause :duration .5)
        (note :duration 2 :pitch (ramp 61 60))))))

(s (glissando-example) (stretch (glissando-example) 2))
```

Figure 3a. Definition of a ramp time function and a musical object using it.

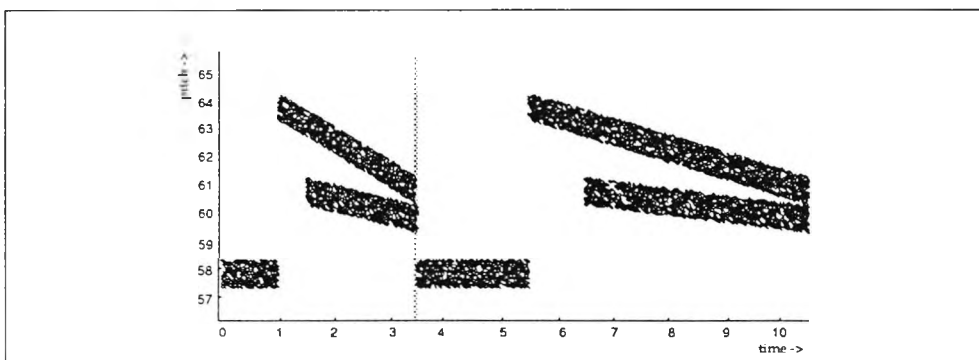


Figure 3b. Glissandi that extend when stretched in time given in Figure 3a.

Now let us construct, with the same means, a vibrato function, parametrized by the fundamental pitch it is to be applied to, and the frequency and depth of the vibrato itself. It only depends on the actual time elapsed during the musical object: the multiplication of duration and progress. Now the application of the vibrato function to notes of different duration will not alter the vibrato rate, nor will the stretch transformation applied to the compound musical object (see Figure 4b).

If we make a similar sinusoidal glissando function, expressed in terms of relative time (progress), a stretch of the musical object will slow-down the rate (see Figure 4c).

Absolute time can also be used to make time functions that are not influenced at all when they are applied to musical objects with a different duration. An ornament could have such character; a sinusoidal ornament will keep its absolute timing when stretched (see Figure 4d).

```
(defun sine-oscillator (offset frequency depth)
  #'(lambda (start duration progress)
    (+ offset
      (* depth
        (sin (* 2 pi duration progress frequency))))))

(defun sine-glissando (key depth)
  #'(lambda (start duration progress)
    (+ key
      (* depth (sin (* 2 pi progress))))))

(defun sine-ornament (key count)
  #'(lambda (start duration progress)
    &aux (relative-time (* duration progress))
    (+ key
      (if (< relative-time count)
        (sin (* 2 pi relative-time))
        0))))

(defun vibrato-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sine-oscillator 64 1 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sine-oscillator 61 1 1))))))

(defun glissando-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sine-glissando 64 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sine-glissando 61 1 1))))))

(defun ornament-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sine-ornament 64 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sine-ornament 61 1 1))))))

(s (vibrato-example) (stretch (vibrato-example) 2)) ; Figure 4b
(s (glissando-example) (stretch (glissando-example) 2)) ; Figure 4c
(s (ornament-example) (stretch (ornament-example) 2)) ; Figure 4d
```

Figure 4a. Definition of musical objects using vibrati, glissandi and ornaments.

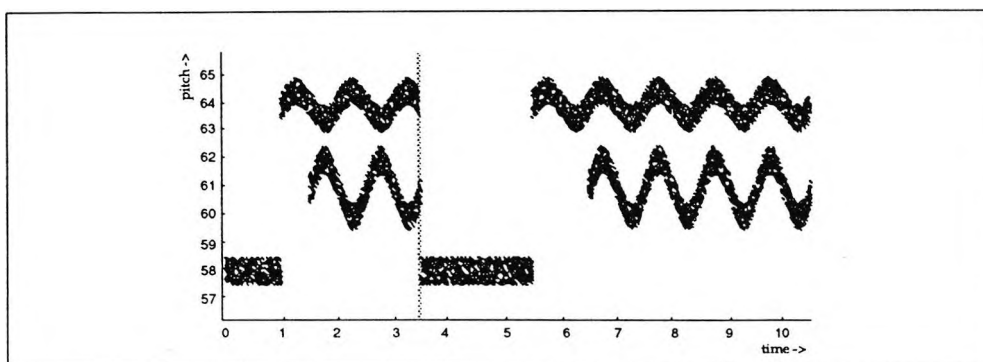


Figure 4b. Vibrati that elongate when stretched in time.

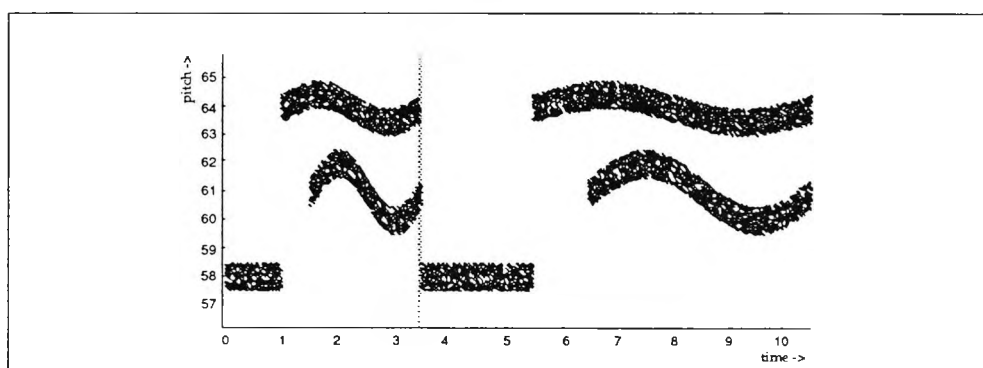


Figure 4c. Sinusoidal glissandi that extend when stretched in time.

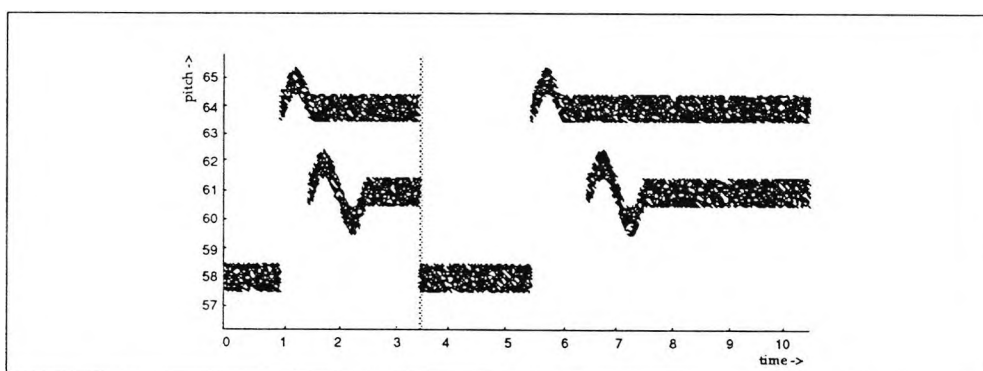


Figure 4d. Sinusoidal ornaments not affected when stretched in time.

The abstract vibrato, glissando and ornament time functions can be depicted nicely in a three dimensional surface plot, as is shown in Figure 5. Relative time (duration * progress) and the duration are used as dependent variables. For any note duration, the actual time function used will be a cross-section of these surfaces.

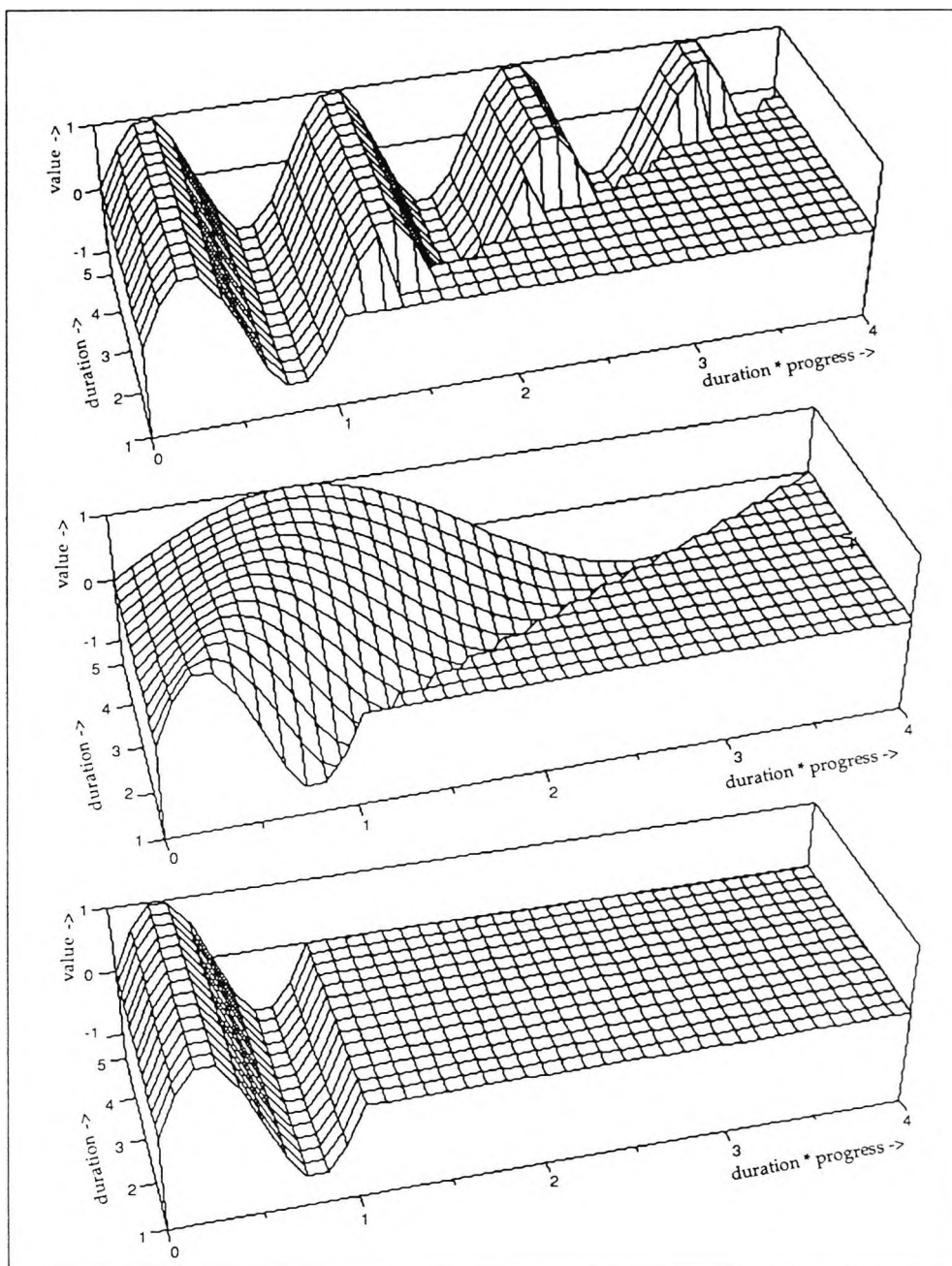


Figure 5. Surfaces representing a) vibrato, b) glissando, and c) ornament time functions as a function of duration and relative time.

Using the absolute start time parameter enables full control over timing with respect to a global clock. This can be used to specify the phase of a vibrato among parallel notes, such that they can be synchronized as is shown in Figure 6 (compare with Figure 4b).

```

(defun sync-oscillator (offset frequency depth &optional (phase 0))
  #'(lambda (start duration progress)
    (+ offset
      (* depth
        (sin (* 2 pi (+ phase
                      (* start frequency)
                      (* duration progress frequency))))))))

(defun synchronized-vibrato-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sync-oscillator 64 1 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sync-oscillator 61 1 1)))))

(s (synchronized-vibrato-example)
  (stretch (synchronized-vibrato-example) 2))

```

Figure 6a. Example of a musical object using a synchronized vibrato.

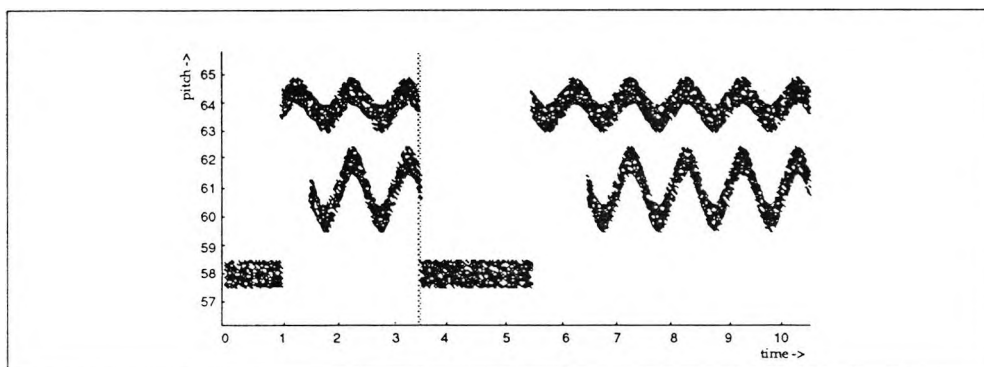


Figure 6b. Graphical notation of the musical object given in Figure 6a.

This finalizes the examples of the use of time functions with multiple parameters as control functions for individual basic objects. It shows how problems of synchronization and time modification are elegantly evaded by lifting the concept of a time function to a more abstract level. Of course the control functions used are rudimentary in their musical value as much more elaborate envelopes are needed, but they can all be based on the same idea and a further section will show how simple time functions can be combined into complex ones. First we want to tackle the problem of time functions extending over a collection of several musical objects.

Control over compound objects

In composition the use of time dependent control specified over a collection of musical objects is abundant. The most simple example specifies the same control information to be applied to each basic object. Naming a control function (as done

with the `let` local binding construct of Common Lisp) is a natural way (see Figure 7).

```
(defun envelope-example ()
  (let ((envelope (ramp 0 1)))
    (s (note :duration 1 :pitch 58 :amplitude envelope)
      (p (note :duration 2.5 :pitch 64 :amplitude envelope)
        (s (pause :duration .5)
          (note :duration 2 :pitch 61 :amplitude envelope))))))
  (s (envelope-example) (stretch (envelope-example) 2)))
```

Figure 7a. Example of applying the same local envelope function to the individual note objects.

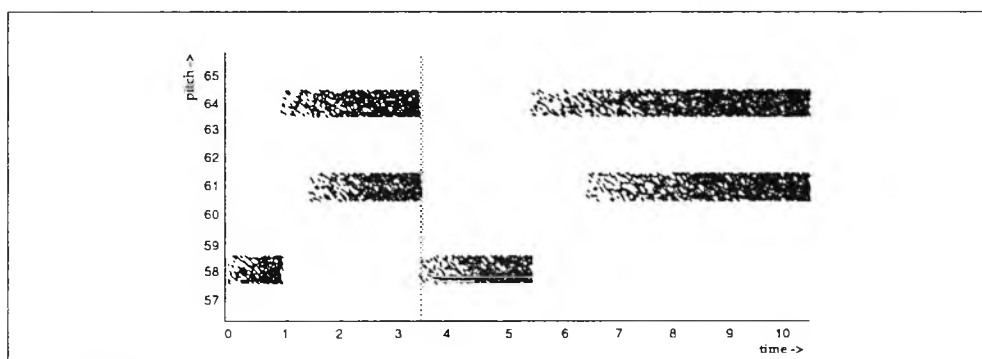


Figure 7b. Graphical notation of the musical object given in Figure 8a.

A different method has to be used to pass time functions evolving over a compound musical object, to each basic musical object. An example of such a construct is a crescendo from a certain loudness level to another, starting at the start of the musical structure it is applied to, and extending over its total duration. We need to introduce a new construct in the language to enable the passing of information from collections of musical objects to such envelopes. It follows the same syntax as the `let` construct but modifies the time functions bound such that they will behave appropriately. In Figure 8 the definition of a crescendo is shown using the same ramp function and the same musical structure as used in Figure 7.

```

(defun crescendo-example ()
  (let-time-fun-over-compound ((crescendo (ramp 0 1)))
    (s (note :duration 1 :pitch 58 :amplitude crescendo)
      (p (note :duration 2.5 :pitch 64 :amplitude crescendo)
        (s (pause :duration .5)
          (note :duration 2 :pitch 61 :amplitude crescendo))))))
    (s (crescendo-example) (stretch (crescendo-example) 2))

```

Figure 8a. Example of applying a global crescendo function to the individual note objects.

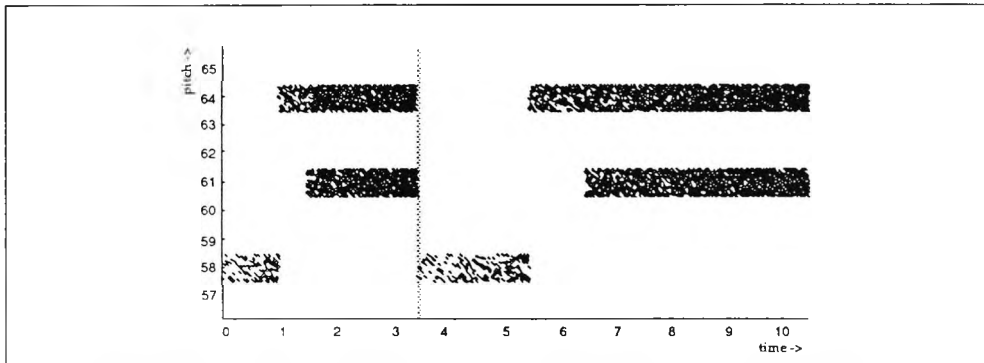


Figure 8b. Graphical notation of the musical object given in Figure 8a.

Time function composition

Building a comprehensive set of musically useful time functions can best be done by supplying some simple, basic time functions and some ways of building complex ones by transforming and combining them. The function `time-fun-compose` is one of the higher-order functions that can be thought of, that supports this. It generalizes any operation to the corresponding combination of the results of time functions. The example in Figure 9 shows both an additive combination of an oscillator and a ramp time function for pitch, and one using a multiplicative combination for the amplitude parameter. An object oriented approach here, packaging time functions in their own class and overloading the standard arithmetic operations for them, will of course simplify the syntax.

```

(defun compose-example ()
  (note :duration 7
        :pitch (time-fun-compose #'(oscillator 58 1 1) (ramp 5 0))
        :amplitude (time-fun-compose #'(oscillator .5 1 .5)
                                       (ramp 1 0))))

(s (compose-example) (stretch (compose-example) .5))

```

Figure 9a. Example of combining time functions

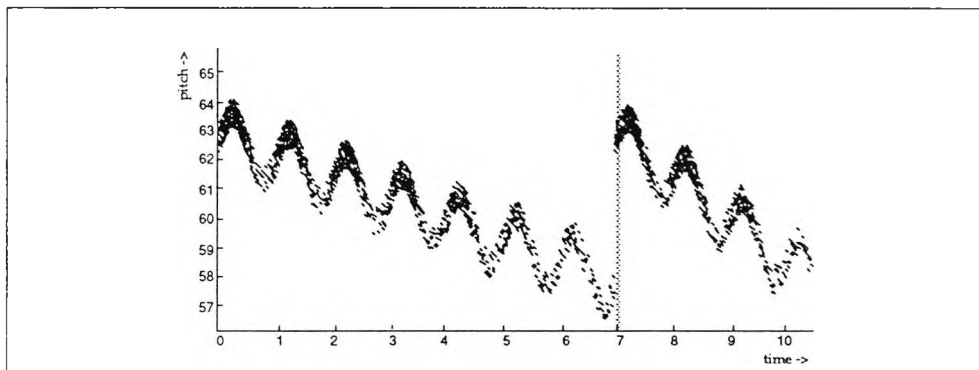


Figure 9b. Graphical notation of the musical object given in Figure 9a.

Another useful combinator is the concatenation of two time functions, with an extra argument expressing the proportion of the duration handled by the first, implying the remaining time for the second one. In Figure 10 the concatenation of two ramp envelopes is shown, one used locally for the pitch, the other used globally for the amplitude parameters.

```

(defun crescendo-decrescendo-example ()
  (let-time-funcs-over-compound
    ((cresc-decresc (time-fun-concatenate (ramp 0 1) (ramp 1 0) .2)))
    (let ((glissando
            (time-fun-concatenate (ramp 58 59) (ramp 59 58) .8)))
      (s (note :duration 1 :pitch glissando :amplitude cresc-decresc)
        (p (note :duration 2.5 :pitch 64 :amplitude cresc-decresc)
          (s (pause :duration .5)
            (note :duration 2
                  :pitch 61
                  :amplitude cresc-decresc))))))

  (s (crescendo-decrescendo-example)
    (stretch (crescendo-decrescendo-example) 2))

```

Figure 10a. Example of concatenating time functions

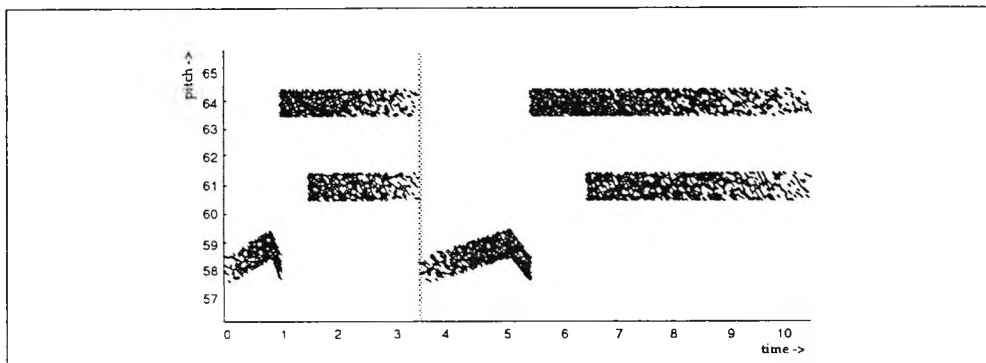


Figure 10b. Concatenations of a crescendo and a decrescendo, and up- and downwards glissandi.

An ever richer world of possibilities opens up when time functions accept time functions as arguments, their parameters may then change over time as well. In the example we use a sine oscillator that changes its frequency over time, controlled by a ramp time function. A new definition of an oscillator, that takes functional arguments, can easily be constructed. In Figure 11a such a sine oscillator time function is defined (it uses the function `time-funcall`, and the function `make-sine` that supplies a sine function that remembers its state over time).

It has to be stressed here again that these combinations of time functions preserve -in a compound way- the different ways in which their constituent components deal with time. For instance, the composition of a glissando and a vibrato, as in Figure 9b, can still be stretched in time consistently: the vibrato gaining cycles at the same rate and the glissando slowing down. This composibility of behavior is an important characteristic of this solution.

```

(defun sine-osc (frequency)
  (let ((sine (make-sine)))
    #'(lambda (start duration progress)
        &aux (time (+ start (* duration progress))))
      (funcall sine time
        (time-funcall frequency start duration progress)))))

(defun new-oscillator (offset frequency depth)
  (time-fun-compose #' + offset
    (time-fun-compose #' * depth
      (sine-osc frequency))))

(defun new-vibrato-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5
      :pitch (new-oscillator 64 2 (ramp 0 1)))
      (s (pause :duration .5)
        (note :duration 2
          :pitch (new-oscillator 61 (ramp 0 1) 1)
          :amplitude (new-oscillator .5 (ramp 0 1) .5))))))
  (s (new-vibrato-example) (stretch (new-vibrato-example) 2)))

```

Figure 11a. Example using an oscillator taking functional arguments.

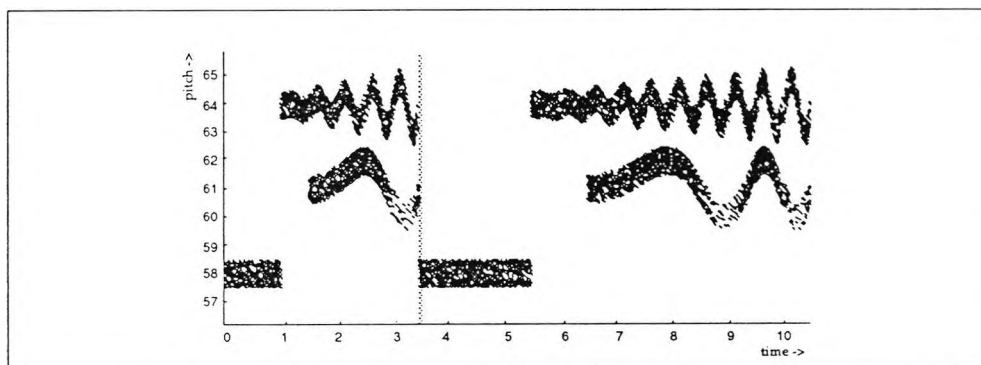


Figure 11b. Graphical notation of the musical object given in Figure 11a.

Specification by means of transformation

Finally, the reader might wonder how transformations on a complex musical object can be taken care of. Because of the abstractions chosen, this comes with little extra effort (in contrast to other systems; see closing remarks in Dannenberg, 1989 and Rahn, 1990). Transformations are just another way of specifying complex musical objects. A set of transformations and their equivalents are shown below.

```

(stretch
  (amplitude
    (s (note) (transpose (note) 2))
    (ramp 1 0))
  2)
=
(let-time-funs-over-compound ((envelope (ramp 1 0)))
  (s (note :duration 2 pitch 60 :amplitude envelope)
    (note :duration 2 pitch 62 :amplitude envelope)))

```

Of course, a free mix of direct specification and transformation can be used. In Figure 12 two examples of simple transformations are given. An attribute-transform function supports these kinds of transformations on the attributes of the notes. It takes, besides the musical object applied to, a keyword identifying an attribute, a time function or constant, and an operator used in combining the results of the time function with the time functions or constants found in the musical object.

```

(defun transpose (object interval)
  (attribute-transform :pitch interval #' + object))

(defun amplitude (object amplitude)
  (attribute-transform :amplitude amplitude #' * object))

(s (transpose (new-vibrato-example) -1)
  (amplitude (stretch (new-vibrato-example) 2)
    (ramp 1 0)))

```

Figure 12a. Examples of attribute transformations.

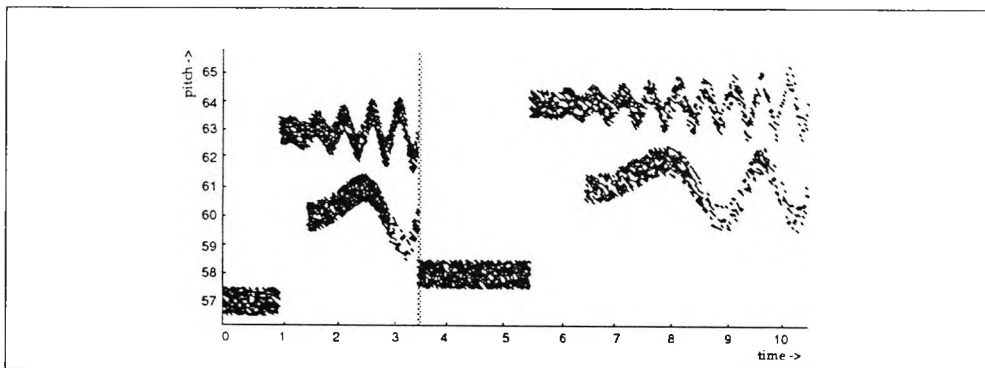


Figure 12b. Graphical notation of the musical object given in Figure 12a.

MICROWORLD

To enable the reader to check and experiment with these ideas, a rudimentary implementation of a language with these capabilities is given in the Appendix, the full implementation of which will be part of the COCO composition system.

The basic and compound musical objects are defined procedurally. Because at the time of their call their context is still unknown (i.e. their start time) they will deliver functions (called event-list generators) of this context that will, when given a start time and a scale factor, return an event-list together with its end time. A draw procedure is available to transform such musical objects into the graphical representation that was used in drawing the figures. Low-level draw routines for the users' window system or plotter, implementing the `draw-air-brush` primitive still have to be supplied. The incorporation of a play primitive to sort event-lists and send data to a MIDI driver is left as an exercise to the reader. Note that this is a non-trivial task since some technical tricks (e.g. allocating notes to different MIDI channels) must be used to allow continuous control of the individual note parameters.

EXTENSIONS

Articulation

Some parameters are not continuously variable by nature (like the articulation of a note: the proportion of its duration that it is actually sounding), but they can be modelled well by using continuously variable time functions extending over compound musical objects, sampled once automatically by the interpretive system at the start time (as in Dannenberg, 1989). It is even possible to supply this information as an extra argument to all time functions (on the risk of becoming circular: articulation time functions themselves should not be allowed to use the articulation parameter). This elegantly solves the specification of, for example, different attack, decay and release sections of envelope functions in terms of an articulation factor.

Real-time control

There is no reason why this approach could not be used in real-time control. If it is possible in the host system to specify the start of a musical object (e.g. a note-on) without its end, then the set of parameters passed to time functions has to be adapted (e.g. to absolute start time, and absolute time elapsed after the start). Time functions can naturally depend upon incoming (real time) parameters if their evaluation is postponed to the last possible moment. Note that in that case time functions are not strictly functional any more, reading control signals from input ports.

Rubato functions and time maps

It might be possible to add flexibility to the common rubato functions and nested time maps used for composition systems (Jaffe, 1985; Desain & Honing, 1991) by supplying them with multiple time arguments as well. To avoid circularity a division in score and performance time has to be made. This could then be a basis for meaningful specifications and transformations of expressive timing. The possibilities have yet to be investigated.

ADVANTAGES OF "TIME FUNCTIONS OF MULTIPLE TIMES"

The proposed approach has three main advantages. Firstly, for the human user the language is transparent, and no unforeseen interactions or side effects take place. Secondly, the musical objects, time functions, and transformations on musical objects are orthogonal constructs. They serve as a solid and extendible basis for further design, independent of the host language or the composition system. Finally, from the hardware perspective this approach has the distinct advantage of being easy to adapt to run on parallel architectures: each note can be handled by a different processor, without the need for information passing between them (see Walker (unpublished) for the argument that note-based parallelism is the most promising distribution of labour for most parallel architectures).

ACKNOWLEDGEMENTS

Thanks to Stephen Pope for the enlightening discussions on music representation during his visit at the Center for Knowledge Technology in the summer of 1990, and Shaun Stevens for his corrections on an earlier version of this text. Roger Dannenberg deserves special thanks for his very useful comments.

REFERENCES

- Abbott, C. (1981) The 4CED Program. Computer Music Journal 5(1). Reprinted in Roads (1989).
- Anderson, D. A. & R. Kuivila. (1989) Continuous Abstractions for Discrete Event Languages. Computer Music Journal 13(3).
- Balaban, M. (1989) Music Structures: A Temporal-Hierarchical Representation for Music. Musikometrika, Vol. 2.
- Berg, P. (1979) Pile, A Language for Sound Synthesis. Computer Music Journal 3(1). Reprinted in Roads & Strawn (1985).
- Buxton, W., R. Sniderman, W. Reeves, S. Patel & R. Baecker. (1978) The Use of Hierarchy and Instance in a Data Structure for Computer Music. Computer Music Journal 2(4). Reprinted in Roads & Strawn (1985).
- Dannenberg, R. (1989) The CANON Score Language. Computer Music Journal 13(1).
- Dannenberg, R., L. M. Dyer, G. E. Garnett, S. T. Pope, & C. Roads. (1989) Position papers. In Proceedings of the 1989 International Computer Music Conference. San Francisco: Computer Music Association.
- Dannenberg, R., P. McAvinney & D. Rubine. (1986) Artic: A Functional Language for Real-time Systems. Computer Music Journal 10(4).
- Desain, P. & H. Honing. (1988) LOCO: A Composition Microworld in Logo. Computer Music Journal 12(3).
- Desain, P. & H. Honing. (1991). Tempo Curves Considered Harmful. In "Music and Time", edited by J.D. Kramer. Contemporary Music Review. London: Harwood Press. (forthcoming)
- Desain, P. & H. Honing. (in preparation) Must "Real-time" Equal "No-idea-of-time" in Composition Systems? Research Report. Utrecht: Center for Knowledge Technology.
- Desain, P. (1990) Lisp as a second language. Perspectives of New Music 28(1).
- Hiller, L., A. Leal & R. A. Baker. (1966) Revised MUSICOMP manual. Technical Report 13. University of Illinois, School of Music, Experimental Music Studio.
- Honing, H. (1991). Issues in the Representation of Time and Structure in Music. In Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross. Contemporary Music Review. London: Harwood Press. (forthcoming)
- Huron, D. (1990) Design Principles in Computer Based Music Representation. In Computer Representations and Models in Music, edited by A. Marsden and A. Pople. London: Academic Press.
- Jaffe, D. (1985) Ensemble timing in Computer Music. Computer Music Journal 9(4).

- Koenig, G. M. (1970) Project 2. Computer programme for calculation of musical structure variants. Electronic Music Reports 3. Utrecht: Institute of Sonology.
- Loy, G. (1985) Musicians Make a Standard: The MIDI Phenomenon. Computer Music Journal 9(4). Reprinted in Roads (1989).
- Loy, G. (1988) Composing with Computers - A Survey of Some Compositional Formalisms and Music Programming Languages. In Current Directions in Computer Music Research, edited by M. V. Mathews & J. R. Pierce. Cambridge, Mass.: MIT Press.
- Mathews, M. V. & F. R. Moore. (1970) GROOVE: A Program to Compose, Store and Edit Functions of Time. Communications of the ACM 13(12)
- Mathews, M. V. (1969) The Technology of Computer Music. Cambridge, Mass: MIT Press.
- Pope, S.T. (1987) A Smalltalk-80-based Music Toolkit. Proceedings of the 1987 International Computer Music Conference. San Francisco: Computer Music Association.
- Pope, S.T. (1989) Modeling Musical Structures as EventGenerators. Proceedings of the 1989 International Computer Music Conference. San Francisco: Computer Music Association.
- Puckette, M. (1988) The Patcher. In Proceedings of the 1988 International Computer Music Conference. San Francisco: Computer Music Association.
- Rahn, J. (1990) The Lisp Kernel: A Portable Software Environment for Composition. Computer Music Journal 14(4).
- Roads, C (ed.) (1989). The Music Machine. Cambridge, Mass.: MIT Press.
- Roads, C. & J. Strawn (eds.) (1985). Foundations of Computer Music. Cambridge, Mass.: MIT Press.
- Rodet, X. & P. Cointe. (1984) FORMES: Composition and Scheduling of Processes. Computer Music Journal 8(3). Reprinted in Roads (1989).
- Schottstaedt, B. (1983) PLA: A Composer's Idea of a Language. Computer Music Journal 7(2). Reprinted in Roads (1989).
- Smoliar, S.W. (1980). A Computer Aid for Schenkerian Analysis. Computer Music Journal 4(2).
- Steele, G. (1990) Common LISP: The language (second edition) Bedford, Mass.: Digital Press.
- Walker, W. (1990) Programming models for digital audio sample generation. Unpublished.

APPENDIX

```
;;; TIME FUNCTIONS MICROWORLD
;;; (C)1991, Peter Desain & Henkjan Honing
;;; Part of the COCO composition system
;;; In Common Lisp (uses loop macro)

;;; basic musical objects

(defun note (&key (duration 1) (pitch 60) (amplitude 1))
  "Return an event-list-generator of a note"
  #'(lambda (start factor &aux (stretched-dur (* duration factor)))
    (values (list (list :start start
                        :duration stretched-dur
                        :pitch pitch
                        :amplitude amplitude))
            (+ start stretched-dur))))

(defun pause (&key (duration 1))
  "Return an event-list-generator of a pause"
  #'(lambda (start factor &aux (stretched-dur (* duration factor)))
    (values nil (+ start stretched-dur))))

;;; compound musical objects (time structuring)

(defun s (&rest elements)
  "Return an event-list-generator of a sequential compound musical object"
  #'(lambda (start factor &aux event-list (end start))
    (loop for element in elements
      do (multiple-value-setq (event-list end)
                              (funcall element end factor))
      append event-list into result-list
      finally (return (values result-list end)))))

(defun p (&rest elements)
  "Return an event-list-generator of a parallel compound musical object"
  #'(lambda (start factor &aux event-list end)
    (loop for element in elements
      with event-list and end
      do (multiple-value-setq (event-list end)
                              (funcall element start factor))
      append event-list into result-list
      maximize end into end-time
      finally (return (values result-list end-time)))))

;;; time transformation

(defun stretch (object new-factor)
  "Return an event-list-generator of a stretched musical object"
  #'(lambda (start factor)
    (funcall object start (* factor new-factor))))
```

```

;;; attribute transformations

(defun attribute-transform (keyword attribute-time-fun operator musical-object)
  "Return an event-list-generator of a transformed musical object"
  #'(lambda (start factor)
    (multiple-value-bind (event-list end)
      (funcall musical-object start factor)
      (loop for event in event-list
        do (setf (getf event keyword)
          (time-fun-compose-local-global operator
            (getf event keyword)
            attribute-time-fun
            start
            (- end start))))))
    (values event-list end))))

(defun time-fun-compose-local-global
  (operator local global global-start global-duration)
  "Return time-function
  composed of operator applied to local and global time-fun"
  #'(lambda (start duration progress)
    &aux (global-progress (/ (+ (- start global-start)
      (* progress duration))
      global-duration)))
    (funcall
      operator
      (time-funcall local start duration progress)
      (time-funcall global global-start global-duration global-progress))))

;;; use of time functions over compound musical objects

(defmacro let-time-funs-over-compound (bindings expression)
  "Establish bindings of time-functions over compound object"
  (let* ((start (gensym)) (end (gensym)))
    `(let* (,start ,end
      ,@(loop for binding in bindings
        collect (make-binding binding start end)))
      ,(make-body start end expression))))

(defun make-binding (binding start end)
  "Return binding of time-function using global start and end time"
  '(, (first binding)
    #'(lambda (local-start local-duration local-progress)
      (let ((local-time (+ local-start (* local-duration local-progress))))
        (funcall ,(second binding)
          ,start
          (- ,end ,start)
          (/ (- local-time ,start) (- ,end ,start)))))))

(defun make-body (start end expression)
  "Return an event-list-generator with time-functions over compound object"
  #'(lambda (start factor)
    (multiple-value-bind (event-list end-time)
      (funcall ,expression start factor)
      (setf ,start start ,end end-time)
      (values event-list end-time))))

```

```

;;; time function utilities

(defun time-funcall (time-fun-or-constant start duration progress)
  "Return constant or result of applying time-function to its arguments"
  (if (functionp time-fun-or-constant)
      (funcall time-fun-or-constant start duration progress)
      time-fun-or-constant))

(defun time-fun-compose (operator &rest time-funs)
  "Return time-function composed of operator applied to results of time-funs"
  #'(lambda (start duration progress)
      (apply operator
              (mapcar #'(lambda (time-fun)
                          (time-funcall time-fun start duration progress))
                      time-funs))))

(defun time-fun-concatenate (time-fun-1 time-fun-2 proportion)
  "Return time-function concatenating two time-functions given a proportion"
  #'(lambda (start duration progress)
      (if (<= progress proportion)
          (time-funcall time-fun-1
                        start (* duration proportion) (/ progress proportion))
          (time-funcall time-fun-2
                        (+ start (* duration proportion))
                        (* duration (- 1 proportion))
                        (/ (- progress proportion) (- 1 proportion))))))

(defun make-sine ()
  "Return sine function with state"
  (let ((phase 0) old-time)
    #'(lambda (time frequency)
        (when old-time
          (incf phase (* 2 pi (- time old-time) frequency)))
        (setf old-time time)
        (sin phase))))

;;; graphical score output

(defun draw (musical-object &key (resolution 1/10))
  "Draw a musical object"
  (loop for note in (funcall musical-object 0 1)
        do (apply #'draw-note resolution note)))

(defun draw-note (resolution &key start duration pitch amplitude)
  "Draw a note using the time-function or constant of the attributes"
  (loop as old-pitch-val = (time-funcall pitch start duration 0)
        then pitch-val
        as old-amplitude-val = (time-funcall amplitude start duration 0)
        then amplitude-val
        as old-time = start then time
        as progress = (/ (- time start) duration)
        as pitch-val = (time-funcall pitch start duration progress)
        as amplitude-val = (time-funcall amplitude start duration progress)
        for time from start by resolution to (+ start duration)
        do (draw-air-brush old-time old-pitch-val time pitch-val
                          old-amplitude-val amplitude-val)))

(defun draw-air-brush (x1 y1 x2 y2 shadel shade2)
  ;; draw-air-brush has to be provided by the user,
  ;; draws a diamond shape with vertical left and righthand sides
  ;; (x1,y1) is mid left, left shadel, (x2,y2) is mid right, right shade2
  )

```



```

;;; examples of use (draws pictures as shown in Figure 4b, 4c and 4d)

(defun sine-oscillator (offset frequency depth)
  #'(lambda (start duration progress)
    (+ offset
      (* depth
        (sin (* 2 pi duration progress frequency))))))

(defun sine-glissando (key depth)
  #'(lambda (start duration progress)
    (+ key
      (* depth (sin (* 2 pi progress))))))

(defun sine-ornament (key count)
  #'(lambda (start duration progress)
    &aux (relative-time (* duration progress))
    (+ key
      (if (< relative-time count)
        (sin (* 2 pi relative-time))
        0))))

(defun vibrato-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sine-oscillator 64 1 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sine-oscillator 61 1 1))))))

(defun glissando-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sine-glissando 64 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sine-glissando 61 1 1))))))

(defun ornament-example ()
  (s (note :duration 1 :pitch 58)
    (p (note :duration 2.5 :pitch (sine-ornament 64 .5))
      (s (pause :duration .5)
        (note :duration 2 :pitch (sine-ornament 61 1 1))))))

(draw
  (s (vibrato-example) (stretch (vibrato-example) 2))) ; Figure 4b
(draw
  (s (glissando-example) (stretch (glissando-example) 2))) ; Figure 4c
(draw
  (s (ornament-example) (stretch (ornament-example) 2))) ; Figure 4d

```

TOWARDS A CALCULUS
FOR EXPRESSIVE TIMING IN MUSIC

Peter Desain & Henkjan Honing

JULY 1991

Submitted to Psychology of Music

© copyright 1991, Peter Desain & Henkjan Honing
Center for Knowledge Technology
Lange Viestraat 2b
3511 BK Utrecht
The Netherlands

CONTENTS

Introduction.....1

Overview of the calculus.....3

 Characteristics.....3

 Representation.....5

 Implementation.....6

Musical objects.....8

 Basic musical objects.....8

 Structured musical objects.....9

 Multilateral structures.....9

 Collateral structures (ornamented objects).....10

S, a multilateral successive structure.....11

P, a multilateral simultaneous structure.....12

APPOG, a collateral successive structure.....13

ACCIA, a collateral simultaneous structure.....14

Example of the representation of a musical object.....15

Representing expression.....16

 Expressive tempo.....16

 Expressive asynchrony.....16

 Expressive articulation.....17

Definition of articulation.....17

 Estimate onsets.....18

 Articulation invariance.....18

Expression maps.....18

 Onset timing.....18

 Articulation expression.....20

Operations on expression maps.....21

 Scale maps.....21

 Scaling expressive tempo.....21

 Scaling the expressive tempo of an S section.....22

 Scaling the expressive tempo of an APPOG section.....24

 Scaling expressive asynchrony.....25

 Scaling the expressive asynchrony of a P section.....26

 Scaling the expressive asynchrony of an ACCIA section.....28

 Scaling expressive articulation.....29

 Scaling the expressive articulation of a multilateral section.....30

 Scaling the expressive articulation of a collateral section.....32

 Keeping articulation consistent in the scaling of expressive timing.....33

 Stretch maps.....35

 Interpolate maps.....35

 Transfer maps.....35

Transformations.....35

 Scale timing.....37

 Keeping articulation consistent.....41

 Scale intervoice expression.....41

Conclusion.....43

Acknowledgements.....43

References.....43

Microworld expression calculus.....45

TOWARDS A CALCULUS FOR EXPRESSIVE TIMING IN MUSIC

Peter Desain & Henkjan Honing

Center for Knowledge Technology
Utrecht School of the Arts
Lange Viestraat 2B
NL-3511 BK Utrecht

This paper presents a calculus that enables expressive timing to be transformed on the basis of the structural aspects of the music. Expression within a unit is defined as the deviations of its parts with respect to the norm set by the unit itself. The behaviour of musical material under expressive transformations is determined uniquely by its structural description and the type of expression. Although the calculus separates different kinds of behaviour, it entails no musical knowledge of the transformations themselves and it also does not model music cognition. The algorithmic simplicity of the calculus combined with its elaborate knowledge representation mirrors the common hypothesis that the complex expressive timing profiles found in musical performances can be explained as the product of a small collection of simple rules linked to a relatively complex structure. The calculus (and the program implementing it) will hopefully prove to be a solid basis for formalised theories of music cognition.

INTRODUCTION

In Desain and Honing (in press, a) we argued that a simplistic notion of a tempo curve of a musical performance is a dangerous and harmful theoretical construct. Although the use of a tempo curve to describe time measurements is perfectly sound, the notion itself is often presented as a cognitive or musical concept. And tempo curves do not have any right to exist in those domains. In the above article, this was concluded from the fact that when it is used as a basis of transformations, inevitably the results make no musical sense. The cause of this failure can often be attributed to the lack of structural information in the tempo curve. For example, in changing the overall tempo of a performance, by manipulating the tempo curve alone, all time intervals of equal length between two notes are scaled in the same way. But some notes may constitute a particular kind of ornamentation, whose duration should be more or less unaffected by tempo. As a result the timing of the piece becomes unmusical. And there are many more examples of transformations that cannot be done on isolated tempo curves. Because the article had an essentially negative tone - identifying the problems and their causes - we felt compelled to follow it up with a study of possible solutions.

This paper is an attempt to identify ways in which structural knowledge can be used to enable expression transformations on musical performances that do make musical sense.

In past research we considered expression merely as deviations of attributes of performed notes from their value notated in a score. This definition, however useful in the initial study of expressive timing, soon lost its attractiveness. In general, listeners can appreciate expression in music performance without knowing the score. And a full reconstruction of the score in the form of a mental representation is often impossible. Take for instance the notion of loudness of notes. Should a listener be required to fully reconstruct the dynamic markings in the score before it is possible to appreciate the deviations from this norm as expressive information added by the performer? Such a nonsensical conjecture indeed follows from a rigid definition of expression as deviations from the score. But it is possible to find ways of defining expression on the basis of performance information only. The more so since it became possible to model the quantization of performed note durations into discrete categories (Desain & Honing, 1991), and therefore even the extraction of performed tempo is possible directly from the performance itself.

In this paper we will base expression on the notion of structural units in a working definition: expression within a unit is defined as the deviations of its parts with respect to the norm set by the unit itself. An example might make this more clear. Let's take, for instance, a metrical hierarchy of bars and beats; the expressive tempo within a bar can be defined as the pattern of deviations from the global bar tempo generated by the tempo of each beat. Or, take the loudness of the individual notes of a chord; the dynamic expression within a chord can be defined as the set of deviations from the mean chord loudness by the individual notes. Using this intrinsic definition, expression can be extracted from the performance data itself, taking more global measurements as reference for local ones, assuming that the structural units themselves are known. Thus the structural description of the piece becomes central, both to establish the units which will act as a reference, and to determine its subunits that will act as atomic parts whose internal detail will be ignored. A generalization of this concept can also deal with expression arising from the interplay of two or more voices.

It will be clear by now that any other connotations of the concept of musical expression, its link to human affect and extra-musical indexicality, however interesting, will be ignored here completely.

Before the details of the calculus are presented it might be fitting to give some explanation for undertaking for this work. First of all, we think that the research of expression in music is in need of measurement instruments that can cope with the enormous complexity of performance data and that are much more sophisticated than tempo curves. Some of the proposed transformations can be used as an "auditory microscope" by

exaggerating expression at certain structural levels, like amplifying the timing lead, the melody often has over the accompaniment. Some of the tools presented can be used as “expression scalpels” for trimming away certain kinds of expression that might obscure other phenomena, like removing the tempo deviations within each beat, but holding the timing patterns of the beats themselves invariant. Other tools can “transplant” musical expression from one piece of music to the other, say from a theme to its variation. The availability of this ‘machinery’ will deepen our understanding of the intricacies of music performance expression.

A further motivation is the practical applicability of this work in systems for computer music. Especially the music editors and sequencer programs that are commercially available nowadays which are in need of better ways to treat musical information in musical ways. Expressive timing should not be considered a nasty feature of performed music, as it is in nowadays multi-track recording techniques where tempo, timing and synchronization are treated as technical problems. Instead expressive timing has to be regarded as an integral quality of performed music whereby the performer communicates structural aspects of the music to the listener (Clarke, 1988). We hope that our work can inspire new music software based on this view.

OVERVIEW OF THE CALCULUS

Characteristics

The calculus has the following important characteristics:

The calculus is described here only for different brands of expressive timing. Dynamics could be formalised along the same lines, but for clarity we restrict ourselves to the domain of expressive timing. Other attributes that carry expression, like intonation, vibrato and timbre may require a different treatment.

The types of expression have to be computable to be within reach of this calculus. One must be able to calculate the expression at every level of the structural hierarchy, given the expression of their components (e.g. the timing of a chord must be computable when the timing of the embedded notes is given). One also must be able to state ways to effectively set the expression of the components once the expression of the whole is given (i.e. propagate a shift in timing down the hierarchy, to the basic objects carrying the expression). Types of expression that do not have this characteristic - or are not yet formalised as such - cannot be described.

Both performance and “score” timing of individual notes are clearly defined. Notes require attributes that can be measured more or less directly from the performance data like the note onset time and the offset time. At least the onset time must be clearly

specified, which makes the calculus less appropriate for expressive performances by instruments for which onset times are not so clear cut. Secondly, the metrical note duration (the timing of the note as notated in the score) must also be available as a note attribute - either via quantization or by matching a performance to a known score. These processes are considered preprocessing here. Although the reference to score duration, score onset and score offset times is less appropriate in the context of our definition of expression - we will use this terminology, for lack of better terms.

The "score" timing of rests is clearly defined. Perhaps surprisingly, the rest plays a key role in some transformations. So we assume that it either can be inferred from the performance timing (Longuet-Higgins, 1976 shows a way of doing so), or it is recovered via the matching of a performance and a known score.

All proposed transformations are structure preserving. This means that the calculus is restricted to true expressive transformations: the score timing of the notes is known and fixed, and transformations will leave this and the structural description invariant.

The behaviour of musical material under expressive transformations is determined uniquely by its structural description and the type of expression.

The transformations are defined on a hierarchical structural description uniquely linking all material. Ambiguous structural descriptions (e.g. two or more possible structural descriptions) or incomplete descriptions cannot be dealt with. The obvious need for knowledge representations containing multiple structural descriptions (metrical, phrase, and rhythmical grouping structures, different analysis etc.) is not denied. We just require that such representations be preprocessed to select only one complete structural description. This is not a real restriction since transformations based on different kinds of structural knowledge of the same piece can always be done in sequence. Re-inserting the trimmed structural descriptions into a transformed piece is trivial because the transformations preserve the structure.

Naturally, the higher-level structural description of the piece must be consistent with the performance timing. For example, a structural description of the piece in which two notes are given a certain sequential time order (one after the other) - can only fit a performance in which at least the onset of the corresponding notes obey the same order. The precise rules will be given when the structural descriptions are introduced.

The transformations are defined to apply to a certain level of the hierarchical structural description, ignoring details from lower levels and keeping higher levels invariant. Means to select such a level are assumed. In sophisticated realisations of the calculus

this may entail a match language ("the first bar of the piano solo that begins with a C") or a graphical representation. In this paper we will simply assume that each musical object has a name as attribute and defines a structural level as the set of objects with a certain name.

Although the calculus separates different kinds of behaviour, it entails no musical knowledge of the transformations themselves. Accordingly, the proposed knowledge representation does support for example, arbitrary descriptions of the metrical structure of a piece, but has no knowledge of "the best structural analysis". To give a second example: the proposed knowledge representation does support ways to modify timing (a)synchrony between voices, but it has no knowledge about correct or effective ways of using this in musical performance.

The calculus also does not model cognition. It does not state how, for example, voice-leading helps auditory streaming, how phrase final lengthening beyond a limited range disables rhythm perception, or how structure is communicated by the expressive timing profiles. However, this work constitutes a solid basis for formalised theories about these issues, providing a powerful representation in which they can be expressed.

Representation

Several concepts are used in the calculus:

Musical objects are either of a basic nature or form a structural description of a collection of musical objects. Basic musical objects consist of notes and rests. Notes are the only musical objects that carry the expressive information. Structural descriptions form collections of musical objects. They may describe hierarchical time intervals like metrical-, phrase- or rhythmical grouping, they can group the notes of chords and ornaments together, or form large horizontal slices through the piece, describing the separate voices etc. Mere collections (sets) of objects are too meager a basis for most transformations, therefore, structural descriptions specify the intended relations in time between these objects as well (Honing, 1991). Most transformations can be defined if two orthogonal characteristics of the structural description are given: the temporal nature and the ornamenting quality. The first describes whether a sequence or a parallelism (a so called successive or simultaneous construct) is represented. The second describes whether the musical object is considered an ornament attached to another object or not. Ornaments are shielded from certain modifications and refer to another object for certain attributes. These two binary characteristics result in four concrete types of structural description that will be described in detail later.

Expressive magnitudes are values of expressive measurement on a certain scale. The scales themselves are of course crucial in modeling effective transformations, in cognitive and musical senses. For example, a tempo scale on which a transformation to make something twice as fast actually yields a double perceived tempo is quite useful. But for the sake of simplicity we abstract from the perceptual processes and the instruments that generate the sound, and will just assume simple physical measurements of time and other expressive attributes.

Expression maps describe the expressive patterns of structured musical objects at a certain structural level. They consist of a section for each musical object at that level. A section lists the expressive values for all components of that object. They come in different brands - consistent with the type of musical structure where they were extracted from. Expression maps can be extracted from and applied to musical objects, with possibly a modification of the map in between.

Expression types are sets of procedures to extract a particular type of expression map from a musical object, to impose it on a musical object, and to modify the map. They capture the difference between expressive tempo, asynchrony, and articulation. They may become fairly sophisticated, like a brand of expressive tempo that knows how to keep the articulation of an individual note invariant when the timing of the note onsets is changed.

Modifications are defined as operations on expression maps. They may scale, interpolate, or do any other operation on the map. They are often designed such that certain characteristics are kept invariant, e.g. the total duration of a section while changing the timing of the parts.

Transformations are defined as operations on musical objects. They are often direct generalizations of the expression map modifications - first extracting the map, applying the modification and imposing the modified map. They also handle the selection of the level of structural description on which to apply the transformation. Furthermore, they may have means to maintain consistency among the affected level and other musical material, e.g. making an accompaniment obediently follow the transformation in expressive tempo applied to the melody.

Implementation

Part of the work described in this paper was done in the design of the POCO system (Honing, 1990) for which a scaling operation of expressive timing linked to structural descriptions was implemented. But, in evaluating this rather complex piece of software, better abstractions arose. Especially the design of a set of data structures for music that

capture the differences in behaviour under transformation proved beneficial. Which again illustrated the adage:

"Get your data structures correct first, and the rest of the program will write itself." (David Jones, quoted in Bentley, 1988)

Because the constructs interact heavily, and because it should be easy to add unforeseen new constructs (like a new type of expression), musical objects, expression maps and expression types are implemented as classes in an object-oriented language. In that way it is easy to express modifications and transformations as polymorphic operations that will behave according to the type (the class) of their arguments. The slicing-up of knowledge in classes means answering questions like: which part of the extraction procedure of an expressive tempo map of a sequential musical object is specific for expressive tempo only and should be stated within the expression type; which part only depends on the sequential nature of the musical structure, and should be part of the class for sequential musical objects; and which part describes the creation of an expression map and belongs to that class?

Although a good Object-Oriented Language (we used CLOS, a Lisp-based system) provides one with the programming-constructs needed to express these concepts, the actual process of factoring knowledge into these polymorphic procedures is still a difficult one, especially because during the design of the best structure of the classes - allowing for the most elegant factoring of the procedures - cannot be completely foreseen. This forced us to go through several re-design rounds before the concepts stabilized in their present form.

The following CLOS (Keene, 1989; Steele, 1990) constructs were used heavily in the implementation: multiple inheritance (forming class dependencies that are more complex than simple hierarchies), multi-methods (functions that are polymorphic in more arguments), mix-in type of inheritance (grouping of partial behaviour in an abstract class that must be mixed in with other classes to supply that behaviour to their instances), method combination (providing ways of combining partial descriptions of behaviour of one method for more classes). Together they make it possible to extend the system by adding program code only, instead of rewriting it.

The calculus will be incorporated in POCO. The other tools available in POCO, like score-performance matchers, multiple structural descriptions, storage and retrieval from standard MIDI-files, playback and editors for music text formats etc., will support a comfortable use of the calculus with real performance data. An implementation in the form of the microworld is given in the appendix and aims at conciseness and elegance. Luckily, this goal only occasionally conflicts with computational efficiency.

The following five paragraphs will describe the calculus in more detail. The reader interested in the more general aspects of the calculus is advised to continue reading below Transformations.

MUSICAL OBJECTS

Musical objects come in different brands. Some types are specific enough to describe an object completely (the instantiatable or concrete classes). Other types are used as a descriptive grouping of likewise behaviour (the abstract classes). The types of musical objects and their interrelations are shown in figure 1.

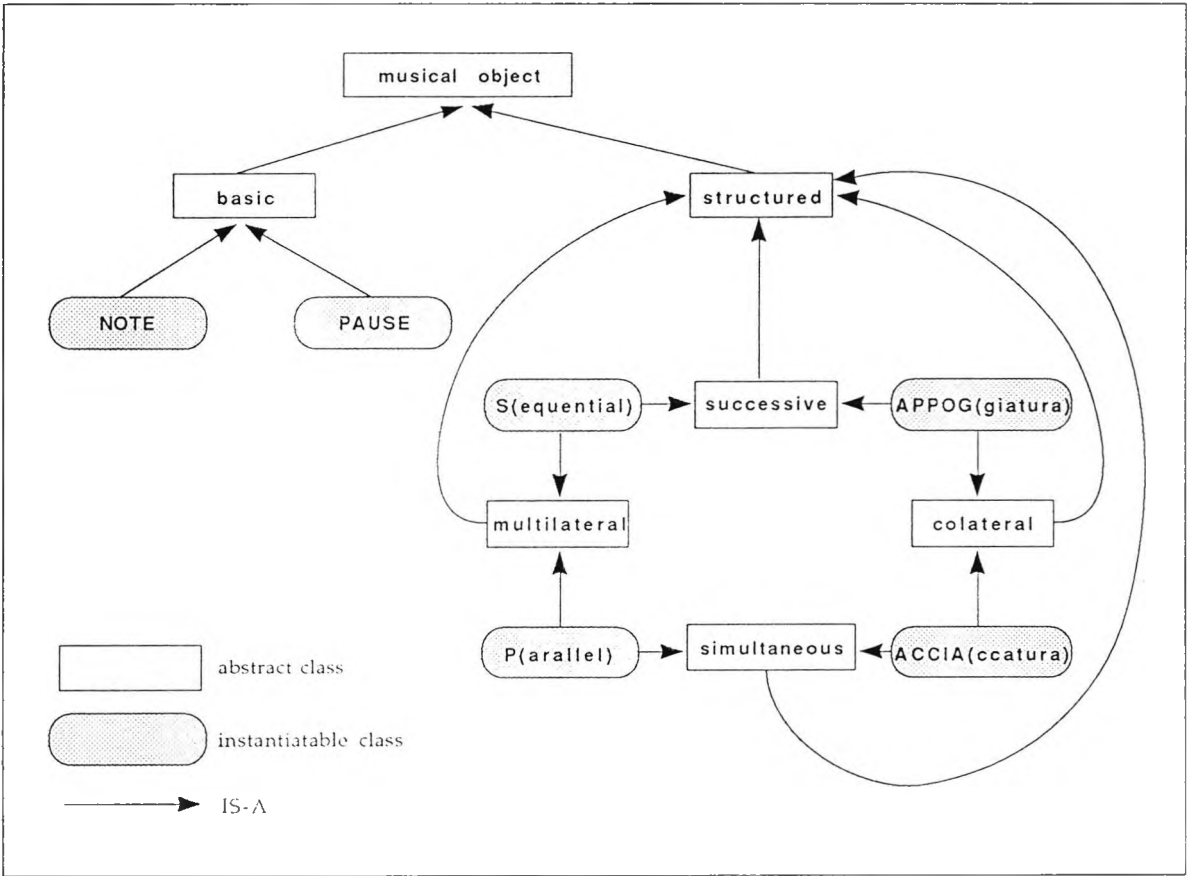


Figure 1. Classes of musical objects and their interrelations.

Basic musical objects

Basic musical objects are notes and rests (In the program we use the word PAUSE to avoid the name clash with the Common Lisp primitive REST). In examples we will use notes with clearly observable onset and offset times (called PERF-ONSET and PERF-OFFSET) measured in ms. from the beginning of the performance. Both notes and rests have as a property a time position in the score (called SCORE-ONSET and SCORE-OFFSET) measured in any kind of (beat)-count (a rational number). These score times are calculated automatically from the supplied score durations of notes and rests via the structural descriptions. This facilitates easy creation of large scores.

Rests are essential and cannot just be ignored, as is done in some low-level representations (e.g. the Midi-file standard). They are central e.g. in dealing with articulation - a short

note followed by a rest behaves differently under transformation than a longer note played in a staccato way.

Structured musical objects

Multilateral structures

In research on music perception and cognition a distinction is often made between successive temporal processes that deal with events occurring one after another, and simultaneous temporal processes that handle events occurring around the same time (e.g. Bregman, 1990; Serafine, 1988). For the first type of events of the expressive means can be rubato - the change of tempo over the sequence. In the second one the expressive means can be chord-spread and asynchrony between voices, both more timbral aspects. These processes work differently in perception. Since we want to imply differences in behaviour mainly by differences in structural description a way should be found in which both these constructs can be represented.

We propose to use for this purpose the simple time structures S and P that functioned well in (Desain & Honing, 1988; Desain, 1990; Desain & Honing, in press, b). If a collection of musical objects is formed such that they occur one after another they are described as a successive structured object named S (for Sequential). If a collection of musical objects occur at the same time they can be collected in a simultaneous structured object called P (for Parallel). These structures serve as a general way to represent a collection together with the temporal relation between the components, as stated in the score. We call the objects *multilateral* because their components are considered to be of equal importance, and are to be treated as such in expressive transformations.

The score times of a structured object and its parts are constrained by consistency rules. They are described separately in frames 1 and 2. These constraints are enforced by specifying only notes and rests with a score duration. The constraints propagate these automatically when a structural description is created and set all score onset and offset times.

In calculating expression, the previous and subsequent context of musical objects is sometimes needed. For instance, consider articulation: possibly defined as the overlap between the sounding parts of a note and the next one, i.e. the time difference between the offset of the note itself and the onset of the "next" note. Besides "next material" a link to "previous material" is foreseen to be needed as well, e.g. in the calculation of local accent patterns. To formalize and generalize this notion of "previous" and "next" material a definition of the left and right context of a musical object is given. This notion also reflects the fact that some expressive values cannot be calculated because some contexts are not available or carry no expression e.g. the tempo of the last note in a piece, or the performance onset of a voice that starts with a rest. Expressive transformations must thus expect to come across missing values in an expression map. The notion of context is

explicitly represented in the program as attributes of the objects themselves. This is possible because the structural description is invariant and so are the contexts. Another possibility would be to represent them implicitly, recovering them by search via a bi-directional part-of link between musical objects. Alternatively, they could be represented tacitly, i.e. supplying them by a general control structure that walks through structured musical objects.

Collateral structures (ornamented objects)

Some musical objects contain components that should maintain a dependency relation to one another. If such a *collateral* pair is transformed, the transformation should be carried out on the main component only, the submissive one obediently following the main component's transformation, but not being transformed itself. An ornamented musical object like a graced note (a note preceded by a grace note), is a good example of a collateral object. For example, in the scaling of the expressive tempo of a melody which contains a graced note, the data on which the expressive transformation is carried out (in this case the performance onset) stems from the main object. The grace note is ignored. When in the actual transformation the graced note pair is stretched or compressed and moved to an other point in time, only the main note will undergo that operation. The ornament will just follow its shift in time.

A second use of this concept is made when the relation of an ornament to its main object, within such a collateral couple, is considered to be expressive, and a potential source of expressive transformations. In this case, the main object stays invariant, and only the ornament undergoes transformation. Take for example the asynchrony between the performance onset of a grace note and the note it is attached to. This time interval can be modified by appropriate means, resulting in local changes of the timing of the grace note - but keeping the timing of the main note invariant.

Collateral (ornamented) objects can again have two kinds of temporal nature: successive or simultaneous. The first one is called APPOG (for *appoggiatura*). It describes a "time-taking" ornament where the ornament is considered to finish when its main object starts (all in terms of score times). The second is called ACCIA (for *acciaccatura*). It can represent a so called "time-less" ornament that is supposed to start at the same time as the object it is attached to. Note that both parts of a collateral pair are musical objects themselves and can have internal structure. The concepts of APPOG and ACCIA ornamented objects are an elaboration of the PRE and POST objects that were introduced in (Desain & Honing, 1988). Consistency rules for score times and context are described in frames 3 and 4.

S, a multilateral successive structure

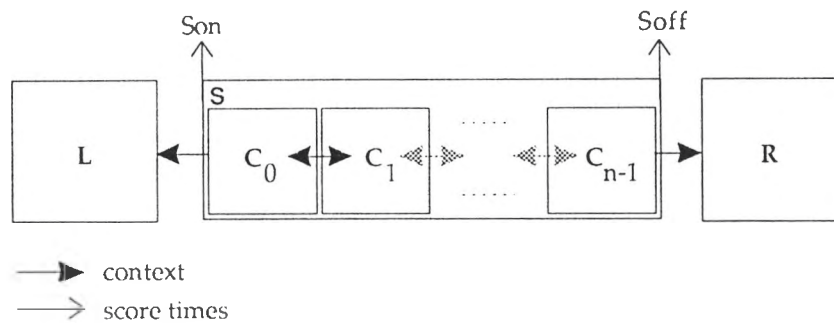


Figure of S object

Consider an S structure of n components C_i with $0 \leq i \leq n-1$.

Assume that component C_i has score onset time So_{n_i} , score offset time So_{ff_i} and that the whole structure has score onset time So_n and score offset time So_{ff} . Then the following must hold:

$$S_{on} = S_{on0}$$

$$\text{Soff} = \text{Soff}_{n-1}$$

$$\text{Soff}_i = \text{Son}_{i+1}, \text{ for } 0 \leq i \leq n-2$$

Assume that component C_i has performance onset time Pon_i and that the whole structure has performance onset time Pon . Then the following must hold:

$$P_{on} = P_{on0}$$

$$Pon_i \leq Pon_{i+1}, \text{ for } 0 \leq i \leq n-2$$

Assume that component C_i has left context L_i and right context R_i and that the whole structure has left context L and right context R . Then the following holds:

$$L = L_0$$

$$R = R_{n-1}$$

$$R_i = C_{i+1}, \text{ for } 0 \leq i \leq n-2$$

$$L_i = C_{i-1}, \text{ for } 1 \leq i \leq n-1$$

Frame 1. Description of a S structure.

P, a multilateral simultaneous structure

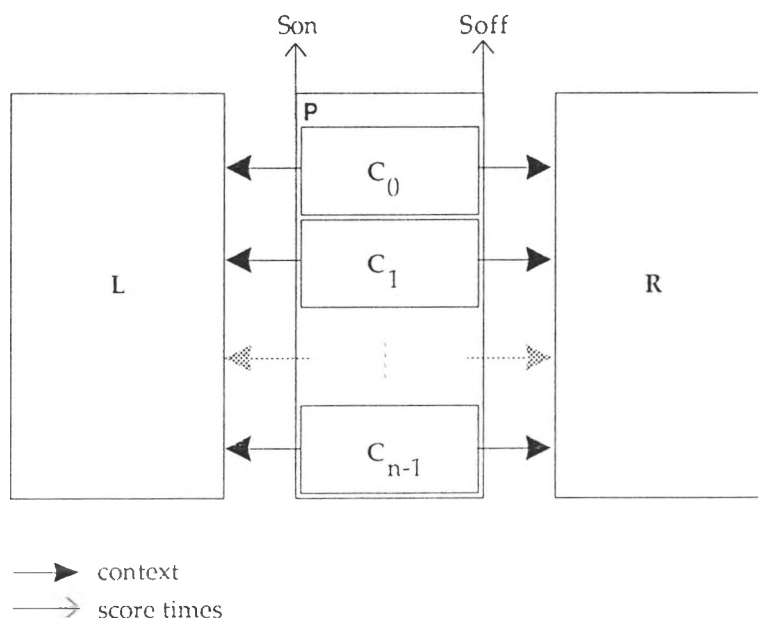


Figure of P object

Consider a P structure of n components C_i with $0 \leq i \leq n-1$.

Assume that component C_i has score onset time Son_i and score offset time $Soff_i$ and that the whole structure has score onset time Son and score offset time $Soff$. Then the following must hold:

$$Son_i = Son, \text{ for } 0 \leq i \leq n-1$$

$$Soff_i = Soff, \text{ for } 0 \leq i \leq n-1$$

Assume that component C_i has performance onset time Pon_i and that the whole structure has performance onset time Pon . Then the following holds:

$$Pon = \text{MIN}_{0 \leq i \leq n-1} Pon_i$$

Assume that component C_i has left context L_i and right context R_i and that the whole structure has left context L and right context R . Then the following holds:

$$L = L_i, \text{ for } 0 \leq i \leq n-1$$

$$R = R_i, \text{ for } 0 \leq i \leq n-1$$

Frame 2. Description of a P structure.

APPOG, a collateral successive structure

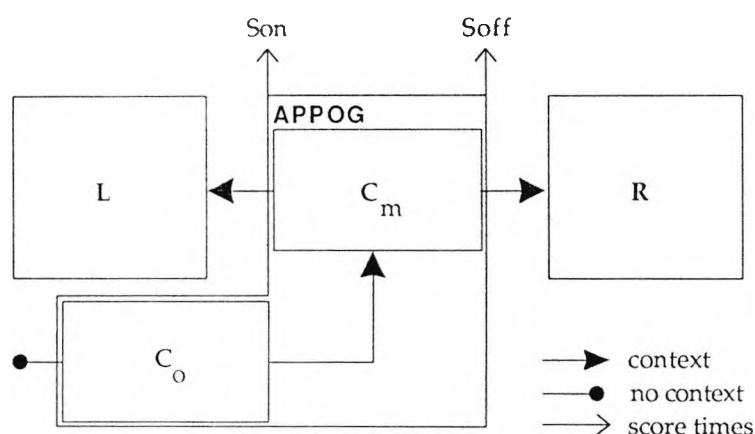


Figure of APPOG object

Consider a APPOG structure of a ornament component C_o and a main component C_m . Assume that component C_o has score onset time Son_o , score offset time $Soff_o$, that component C_m has score onset time Son_m and score offset time $Soff_m$ and that the whole structure has score onset time Son and score offset time $Soff$. Then the following must hold:

$$\text{Son}_m = \text{Son}$$

$$\text{Soff}_m = \text{Soff}$$

$$\text{Soff}_O = \text{Son}_m$$

Assume that component C_o has performance onset time Pon_o , component C_m has performance onset time Pon_m and that the whole structure has performance onset time Pon . Then the following holds:

$$P_{on} = P_{on_{n1}}$$

$$P_{on_0} \leq P_{on_m}$$

Assume that component C_0 has left context L_0 and right context R_0 , component C_m has left context L_m and right context R_m and that the whole structure has left context L and right context R . Then the following holds:

$$L = L_m$$

$$R = R_m$$

$$R_o = C_m$$

$L_0 = \text{undefined}$

Frame 3. Description of an APPOG structure.

ACCIA, a collateral simultaneous structure

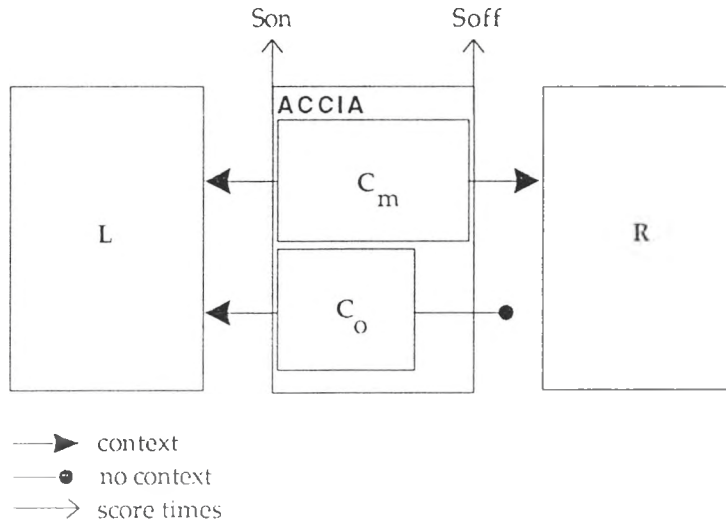


Figure of ACCIA object

Consider a ACCIA structure of a ornament component C_o and a main component C_m . Assume that component C_o has score onset time Son_o , score offset time $Soff_o$, that component C_m has score onset time Son_m and score offset time $Soff_m$ and that the whole structure has score onset time Son and score offset time $Soff$. Then the following must hold:

$$Son_o = Son_m = Son$$

$$Soff_m = Soff$$

Assume that component C_o has performance onset time Pon_o , component C_m has performance onset time Pon_m and that the whole structure has performance onset time Pon . Then the following holds:

$$Pon = Pon_m$$

Assume that component C_o has left context L_o and right context R_o , component C_m has left context L_m and right context R_m and that the whole structure has left context L and right context R . Then the following holds:

$$L = L_m = L_o$$

$$R = R_m$$

$$R_o = \text{undefined}$$

Frame 4. Description of an ACCIA structure.

EXAMPLE OF THE REPRESENTATION OF A MUSICAL OBJECT

In figure 2 a fragment of a score is shown that will serve as a basis for the examples at the end of this article. It is the score of the last bars of the theme of six variations over the duet *Nel cor più non mi sento*, by Ludwig van Beethoven (with some adaptations), which is the same material used to study tempo curves in (Desain & Honing, in press, a). It contains examples of several kinds of musical structure: chords, voices, sequences, bars and beats, phrases and two types of ornaments. Figure 3 shows a graphical notation indicating two structural descriptions: a metrical hierarchy and a separation into voices. The way these structures are specified in Lisp is given in the appendix.

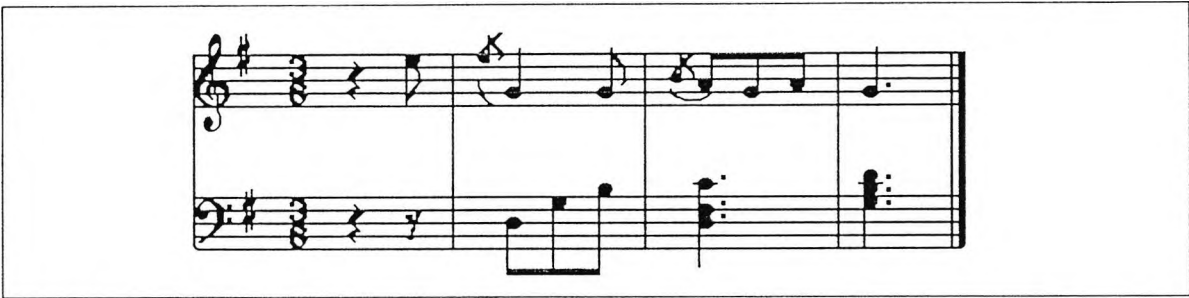


Figure 2. Score of the last bars of the theme of six variations over the duet *Nel cor più non mi sento*, by Ludwig van Beethoven.

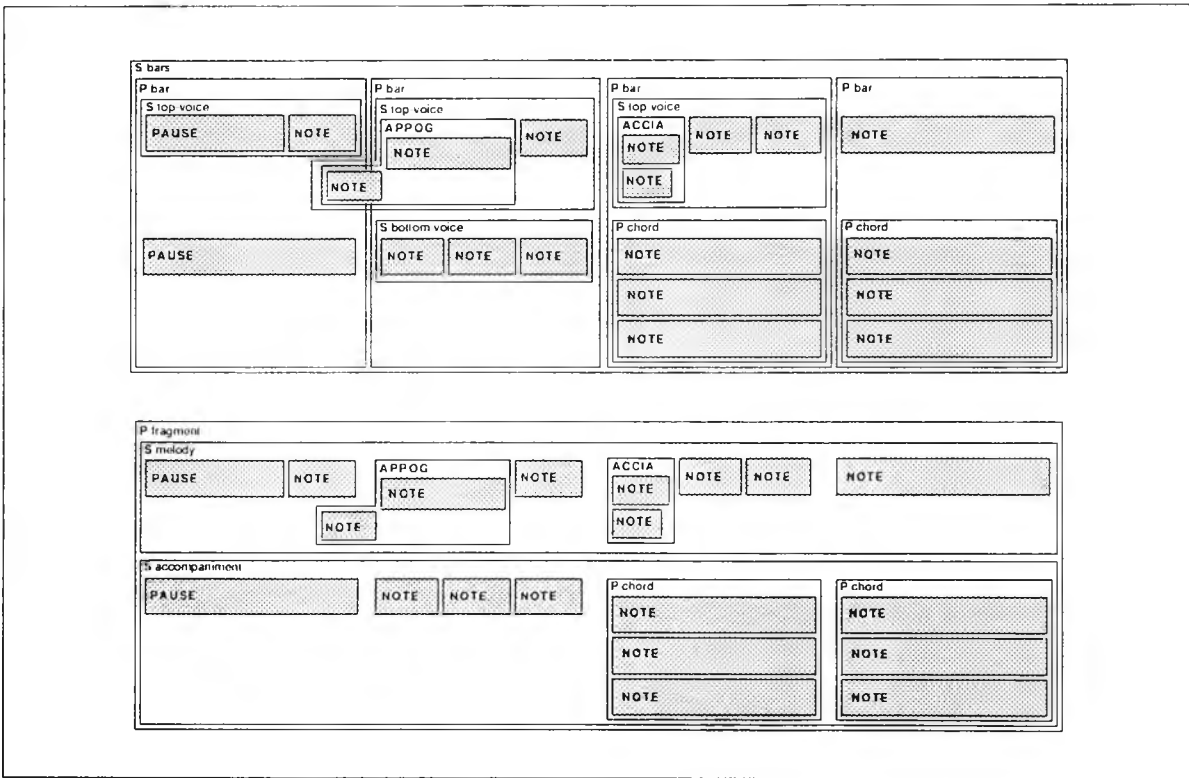


Figure 3. a) Structural description of the metrical hierarchy of the score in figure 2, and b) Structural description of the voices in that piece.

REPRESENTING EXPRESSION

There are three kinds of expressive timing: expressive tempo, expressive asynchrony and expressive articulation. The first two are based on performance onset times only, the third is based on performance onset and offset times (see figure 4).

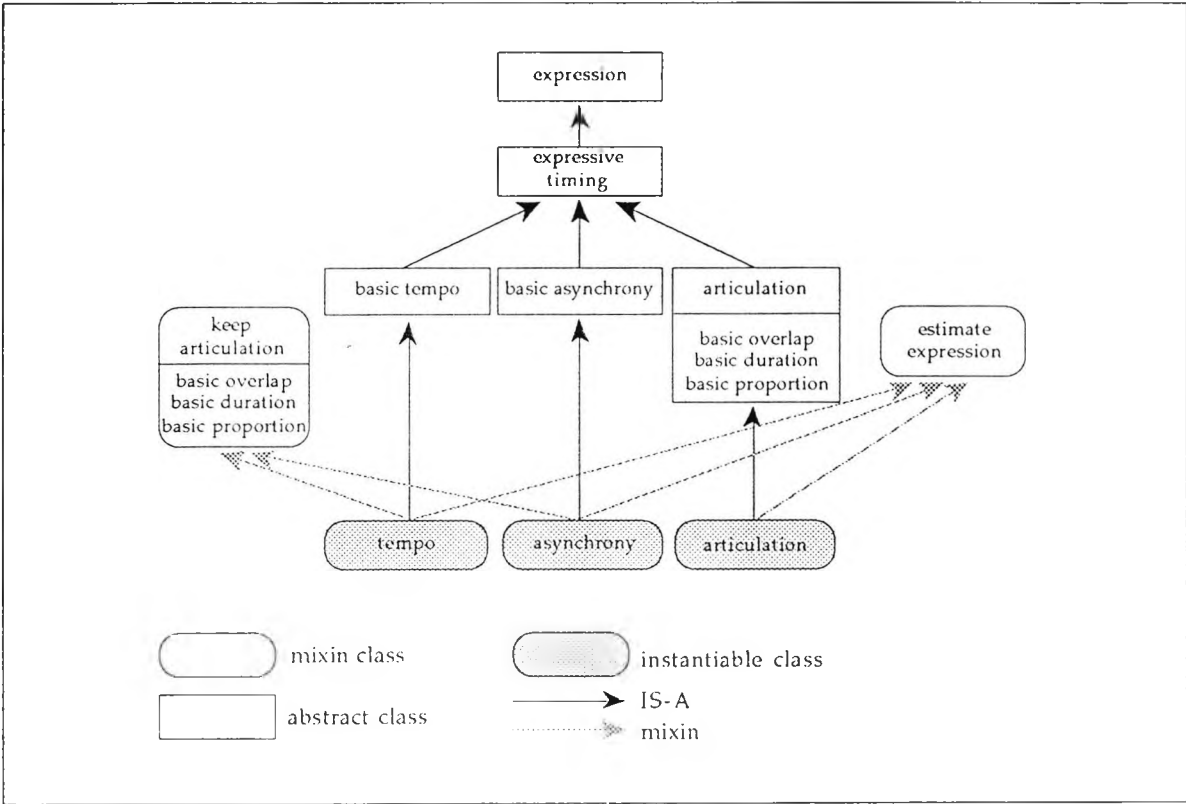


Figure 4. Expression type hierarchy.

One could imagine sophisticated algorithms that calculate the onset of a note and of parallel structures on the basis of their perceptual onset (P-center; see Vos & Rasch, 1981). But for clarity we use a very simple definition of onset times, which was already given in the frames 1 to 4. In that way, all musical objects have performance onset times and so can be used as units on which tempo and asynchrony measures are built.

Expressive tempo

The notion of tempo is relevant only for successive structures. It is defined as the ratio of score duration and performance duration. This velocity-like notion the inverse of the notion of a tempo factor, as is used in the psychology of music literature.

Expressive asynchrony

The notion of asynchrony is relevant only for simultaneous structures. It is defined as the difference of performance onsets. It is thus independent of score times.

Expressive articulation

Expressive articulation uses the performance offsets of individual notes. It simply assumes that they are given. A definition of performance offset of structured musical objects is not needed. Articulation is also independent of score times.

Articulation can be defined in several ways - but it is hard to find a way that will suffice in all circumstances. In the legato range the absolute overlap time of the sounding part of a note and the next one seems a good candidate for an articulation scale. In the staccato range the absolute sounding duration of the note seems the most prominent perceived aspect. In the intermediate range the relative sounding proportion is a good measure. For the moment we cannot do better than to supply these three concepts of articulation expression (overlap-, duration- and proportion-articulation) - leaving it for the user to choose the most appropriate one (see frame 5). For a multilateral structure the expressive articulation value is taken to be the average articulation of its parts. For a collateral structure the expressive articulation value is defined to be the articulation of its main part.

Definition of articulation

Consider a note with performance onset Pon , performance offset $Poff$ and performance onset of its right context Pon_r . There are three alternative definitions of articulation A given:

overlap articulation $A = Poff - Pon_r$

duration articulation $A = Poff - Pon$

proportion articulation $A = \frac{Poff - Pon}{Pon_r - Pon}$

If a multilateral structure with articulation A has components C_i for $0 \leq i \leq n-1$, and C_i has articulation A_i then:

$$A = \text{MEAN}_{0 \leq i \leq n-1} A_i$$

If a collateral structure has articulation A , and its main component C_m has articulation A_m then:

$$A = A_m$$

Frame 5. Definition of articulation expression.

Estimate onsets

Because sometimes the performance onset of missing objects (like the virtual note after the end of the piece) or the performance onset of a rest are needed, we devised a set of procedures that estimates these missing values on the basis of performance onsets that can be found in the context, using a linear interpolation or extrapolation method. The set of procedures forms a mix-in class that can be combined with any expressive timing type enabling that kind of expressive timing to deal - in all operations - with missing values. Estimation is derived from the same structural level as the transformation itself. For example, a transformation on a beat structure in need of a missing expressive value at the end of the piece (cf. the onset the final barline in a score) will be estimated on basis of the two previous beats -not on the basis of any internal detail. In the case of extreme tempo variations, as occur in a final retard, the estimation feature cannot work well. In this case it is better not to use it.

Articulation invariance

When moving the onsets of notes around (e.g. in modifying the performance onsets) it is quite annoying that the articulation of the individual notes also changes - an effect that is very easy to perceive and which may well overshadow subtle modifications of onset timing. Therefore a set of procedures can be mixed-in with expressive tempo and asynchrony. They are given a chance to calculate the articulation of individual notes before onsets are changed and to reinstall it afterwards. This will insure that articulation is kept invariant under transformations of onset timing (see figure 12).

EXPRESSION MAPS

An abstraction of the expression of an object is useful for many operations because it can hide the irrelevant details of the structure and provides a means to transfer expression from one object to another. Therefore expression maps were introduced. They describe expression of musical objects at one level of a structural description. All objects at the level described must have the same structural type. Maps contain a list of sections, one for each of those musical objects. A section lists the expressive values of the components of that musical object. Of course maps may be partial - consisting of several sections with gaps in between, or even have missing values within a section.

Onset timing

The application of a (modified) map of performance onsets on an object works as follows. First, all objects at the indicated level are found, paired with their corresponding sections. Then each section is applied to its object. This means that the components of that object are provided one by one with a new onset from that section.

This setting of onsets is handled differently according on the structural type of the component. If this component is a note, the onset is set directly. For S components the whole structure is stretched between that onset and and the next onset (the onset of the succeeding component). A P component is set to the provided onset, but keeps its internal asynchrony invariant and truncates at the next onset. In the case of a ACCIA component, the main structure is set to the onset, with the ornament following the displacement of the main structure. Finally, for a APPOG component, the main structure is stretched between that onset and the next onset, with the ornament also simply following the displacement of the main object.

Now we have indicated how an expressive timing is applied to components of structured objects - it remains to be shown how such a change propagates when these components again are embedded structured objects themselves. This fairly complex process depends on the type of the embedded structured object and mirrors the decisions given above: S components are stretched, P components are shifted and truncated, and ornaments follow the shift of their main components.

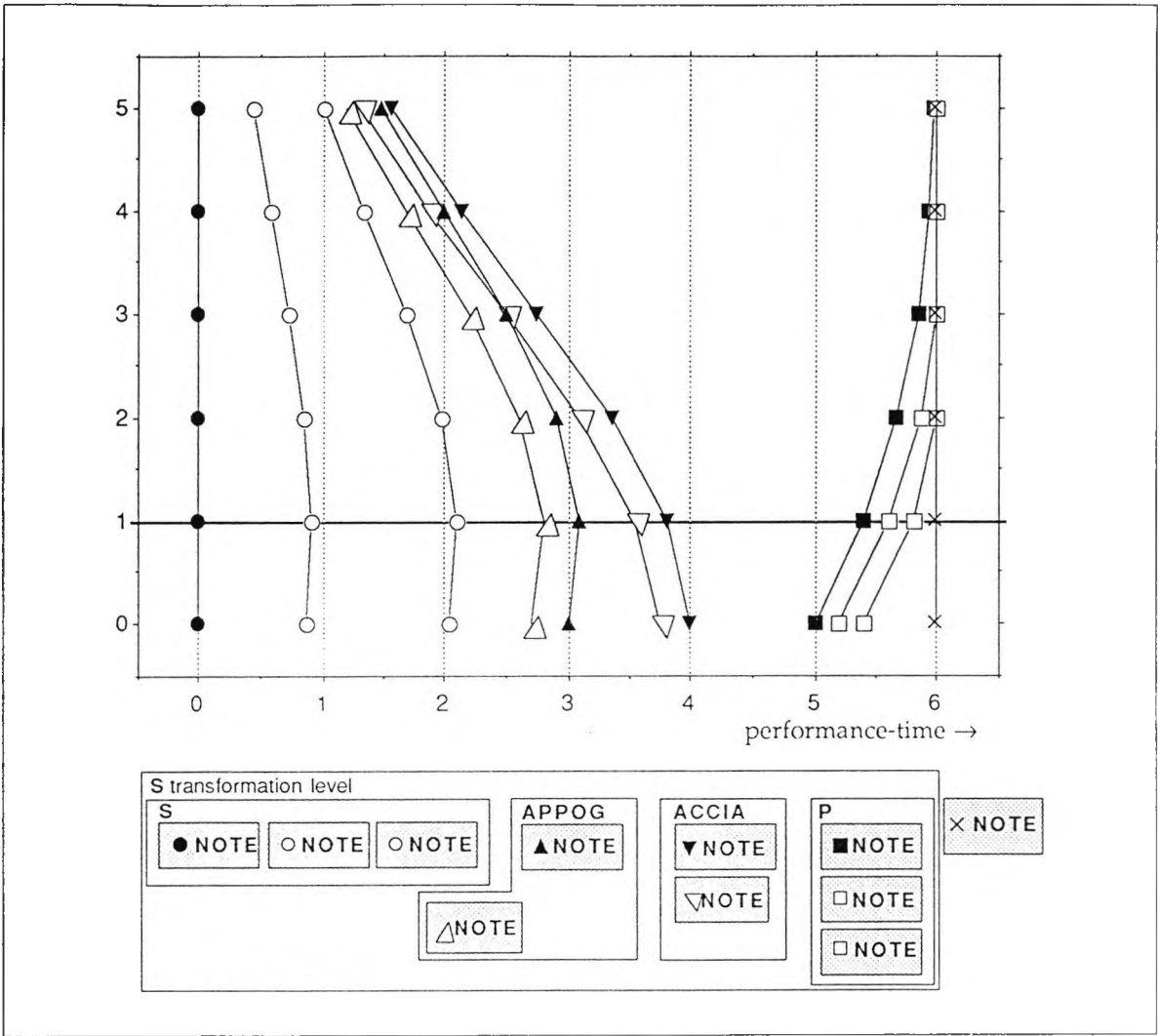


Figure 5. Propagation of change of onset within an S structure for different component types. This figure shows the propagation process for an S structure containing different types of structural components. We assume the components are moved around by an arbitrary transformation, parametrized by a factor. In this figure it is shown how this change is propagated to the internal structure of different kinds of components. The first component is an S structure and the onsets of its internal parts (lines marked with white circles) are stretched along proportionally. The second sub-structure is an APPOG structure and one can see that the onset of its ornament (line marked with upward pointing white triangles) shifts along with the main object. The third sub-structure is an ACCIA structure and the onset of its ornament behaves likewise. Note that the onset of the ornament is allowed to shift freely (line marked with downward pointing white triangles), even the order of notes is allowed to change here. The fourth sub-structure is a P structure and the onset of its components (lines marked with squares) are shifted and truncated at the end (the right context note; line marked with x's).

Articulation expression

In comparison, to set the articulation expression to a structured object is much simpler. When a section of an articulation map is applied to a multilateral or collateral structure the articulation of its components are set to their respective values from the section. The propagation of a (modified) articulation value to a component works as follows. If that component is a note, a new offset is calculated from the articulation value and set directly, taking care to maintain reasonable offset times (e.g. not shifting before its onset). If that component is a multilateral structure, its articulation is calculated (the

mean articulation of its components) and the difference with the required articulation is propagated as an increment to all components. If it is a collateral structure, its articulation is calculated (the articulation of its main component) and the difference with the required articulation is propagated as an increment to both main and ornament components.

OPERATIONS ON EXPRESSION MAPS

Operations on expression maps work section by section. In each section the expression of a structured musical object is represented. The operations delivers a new section to be applied to that object. Care was taken to maintain structural consistency in all operations even in case of extreme parameter values. Of course expression transformations are intended as subtle changes and truncation or extreme normalization should in practice never occur.

Scale maps

Scaling expressive tempo

Scaling tempo is done in an exponential way. Inverse tempi are considered to be related by a scale factor -1; twice as slow is considered to be the mirror image of half as slow. This exponential scaling of expressive tempo mirrors the exponential nature of notated note durations.

Scaling the expressive tempo of an S section

The scaling of the expressive tempo of a multilateral successive structure works as follows. Assume the structure has n components named C_i with $0 \leq i \leq n-1$. Assume component C_i has score onset time Son_i and performance onset time Pon_i . Assume the right context of the structure (and thus the right context of component C_{n-1}) is object C_n . It has score onset Son_n and performance onset time Pon_n . A section of the expressive tempo map of the structure contains all Son_i and Pon_i including Son_n and Pon_n . The scale operation on such a section delivers a new section with performance onsets Pon_i' according to the following rules:

Define the score inter-onset interval ΔSon_i and the performance inter-onset interval ΔPon_i and the local tempo T_i for $0 \leq i \leq n-1$ (a better term would be velocity) as:

$$\Delta Son_i = Son_{i+1} - Son_i$$

$$\Delta Pon_i = Pon_{i+1} - Pon_i$$

$$T_i = \frac{\Delta Son_i}{\Delta Pon_i}$$

This ratio is scaled by an exponential factor f .

$$T_i' = T_i^f$$

Then new raw performance durations $\Delta Pon_i''$ are calculated:

$$\Delta Pon_i'' = \frac{\Delta Son_i}{T_i'}$$

These are re-normalised such that the total performance duration is kept invariant.

$$\Delta Pon_i' = \Delta Pon_i'' \cdot \frac{Pon_n - Pon_0}{\sum_{i=0}^{n-1} \Delta Pon_i''}$$

Starting at the same point, the new performance times are given as:

$$Pon_i' = Pon_0 + \sum_{j=0}^{i-1} \Delta Pon_j'$$

Frame 6. Scaling the expressive tempo an S section.

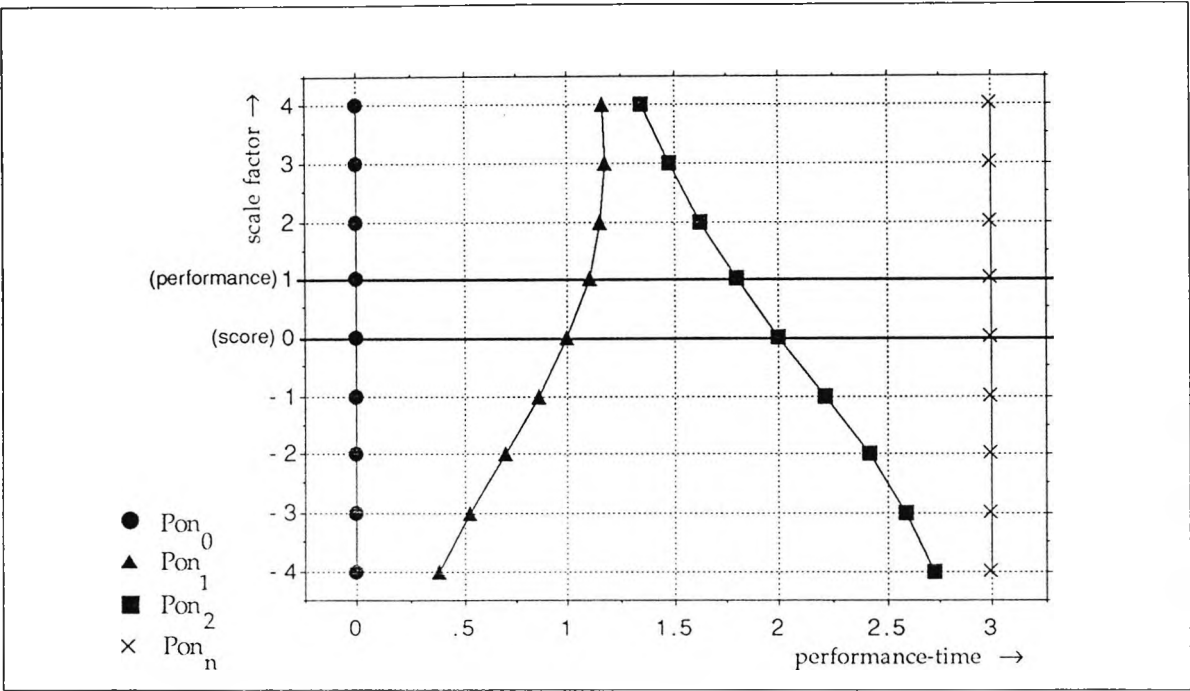


Figure 6. Scaling the expressive tempo of an S section. This process is shown for a specific set of performance onsets Pon_i . In this figure the horizontal axis is the performance time P . On the vertical axis the scale factor f is given. Thus at the horizontal line at scale factor 1 the performance times Pon_i' are shown as markers on the line; they are identical to the original performance times Pon_i . This operation (with scale factor 1) is the identity transformation with respect to the performance timing. At the horizontal line at scale factor 0 the performance times Pon_i' are identical to the score times Son_i (modulo normalization to the total performance duration). This operation (with scale factor 0) effectively removes the expressive timing of the performance. At factor .5 a diminished expressive timing profile will result, and at factor 2 an exaggerated rubato can be obtained. At negative values of the scale factor the expressive profile is inverted: a slower tempo becomes faster and vice versa. At extreme values of the scale factor the note that is played at the slowest tempo in the performance will gain almost the whole performance time interval spanned by the structure, pushing other notes to zero duration. When the performance onset Pon_n is not available, the scale transformation uses Pon_{n-1} instead, and scales the tempo of the section with regard to the onset of the last component in the section - instead of the onset of the right context. This tempo scaling method works well for S constructs with many components and small tempo deviations.

Scaling the expressive tempo of an APPOG section

The scaling of the expressive tempo of a collateral successive structure works as follows. Assume this structure has a main component with score onset time Son_m and performance onset time Pon_m and a preceding ornament component with score onset time Son_o , and performance onset time Pon_o . Assume the right context of the structure (and thus the right context of component C_m) is object C_r . It has score onset Son_r and performance onset time Pon_r . An APPOG time map section contains this score and performance data. The scale operation on such a map delivers a new map with performance onsets according to the following rules:

Define the main and ornament score inter-onset interval $\Delta Son_m, \Delta Son_o$ and the main and ornament performance inter-onset interval $\Delta Pon_m, \Delta Pon_o$ as:

$$\Delta Son_m = Son_r - Son_m$$

$$\Delta Son_o = Son_m - Son_o$$

$$\Delta Pon_m = Pon_r - Pon_m$$

$$\Delta Pon_o = Pon_m - Pon_o$$

The ornament tempo T_o and the main tempo T_m are calculated as:

$$T_o = \frac{\Delta Son_o}{\Delta Pon_o}$$

$$T_m = \frac{\Delta Son_m}{\Delta Pon_m}$$

T_o/m is tempo of the ornament relative to the main tempo. This factor is scaled by an exponential parameter f , and a new ornament tempo T_o' is calculated:

$$T_o/m = \frac{T_o}{T_m}$$

$$T_o' = T_m * T_o/m^f$$

This gives a new performance duration $\Delta P_o'$, which yields the new performance times Pon_m' and Pon_o' :

$$\Delta Pon_o' = \frac{\Delta Son_o}{T_o'}$$

$$Pon_m' = Pon_m$$

$$Pon_o' = Pon_m - \Delta Pon_o'$$

Frame 7. Scaling the expressive tempo of an APPOG section.

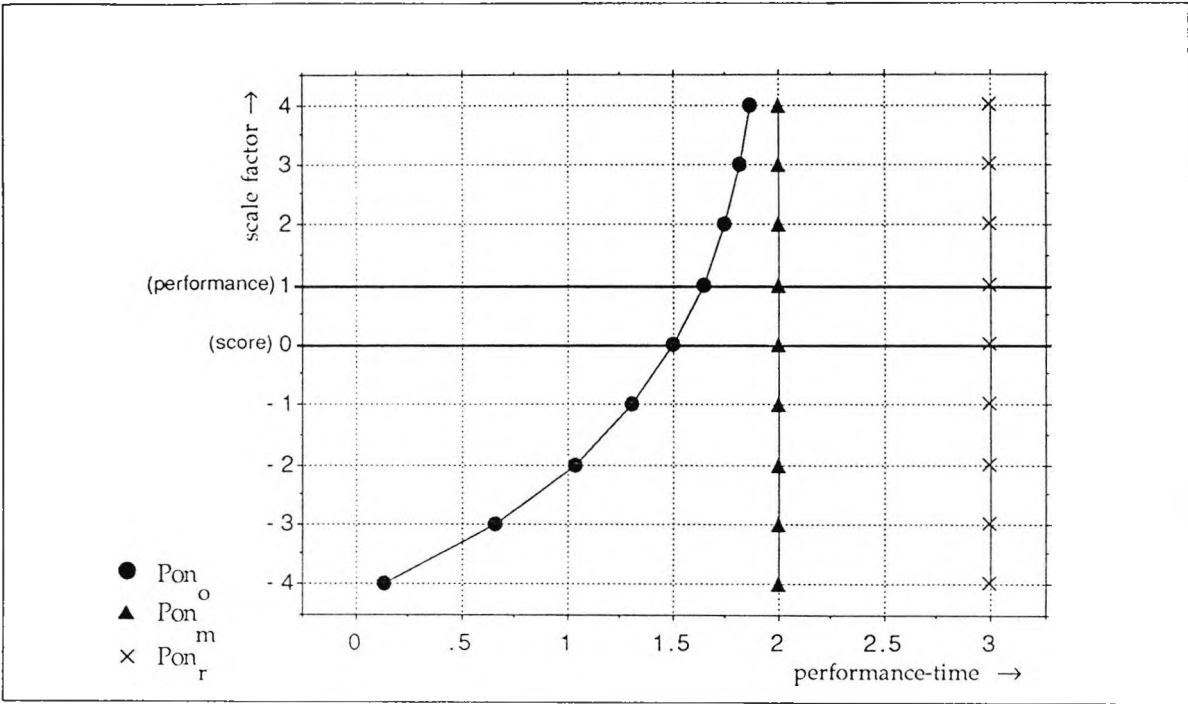


Figure 7. Scaling expressive timing of an APPOG section. This process is shown for a specific set of performance onsets. Note that only the performance timing of the ornament is affected. At scale factor 1 the timing of the ornament is identical to the original timing. At scale factor 0 the ornament is performed at the same tempo as the main object (in this particular example the score duration of the ornament is half that of the main component). This operation (with scale factor 0) effectively removes the expressive way in which the ornament is performed, relative to the main component. At factor .5 a diminished expressive timing effect will result, and at factor 2 an exaggerated effect will be obtained. At negative values of the scale factor the expressive timing is inverted: a performance of the ornament at a lower tempo than the main component becomes one at a faster tempo and vice versa.

Scaling expressive asynchrony

Asynchrony occurs when two or more simultaneous musical objects - prescribed to happen at the same score time - have unequal performance onsets. The differences can be scaled linearly but care has to be taken not to disrupt the timing of higher levels.

Scaling the expressive asynchrony of a P section

The scaling of the expressive asynchrony of a multilateral simultaneous structure works as follows. Assume the structure has n components named C_i with $0 \leq i \leq n-1$. Component C_i has performance onset time Pon_i . Assume the right context of the structure (and thus the right context of all components) has performance onset Pon_n . A parallel time map of the structure contains all Pon_i including Pon_n . The scale operation on such a map delivers a new Pon_i' according to the following rules:

Let the global performance onset Pon and the performance onset asynchronies ΔPon_i be defined as:

$$Pon = \min_{0 \leq i \leq n-1} Pon_i$$

$$\Delta Pon_i = Pon_i - Pon \text{ for } 0 \leq i \leq n-1$$

The asynchronies are scaled by an multiplication factor f :

$$\Delta Pon_i' = \Delta Pon_i * f$$

New performance onsets Pon_i' are calculated, shifting such that the global performance onset is kept invariant ($\min(Pon_i') = \min(Pon_i) = Pon$). The result is truncated such that the onsets never move beyond Pon_n . Of these two safeguards the first applying in case f is negative, the second applying in case f is large compared to the ratio of the asynchronies and performance duration of the whole structure. Together they ensure consistency with higher-level structural descriptions by keeping the components within the bounds of the structure.

$$Pon_i' = \min(Pon_n, Pon + \Delta Pon_i' + \min(\Delta Pon_i'))$$

Frame 8. Scaling the expressive asynchrony of a P section

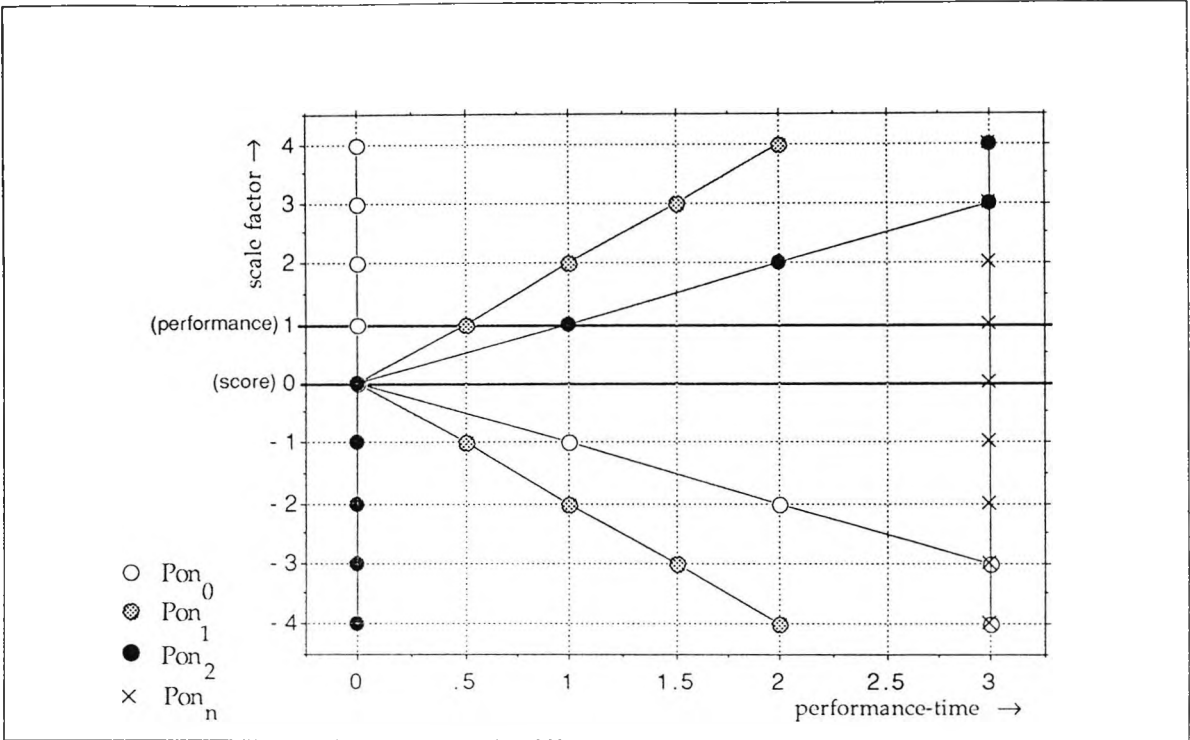


Figure 8. Scaling expressive timing of a P section. This figure shows this process for a specific set of performance times P_i (say a chord performed with some spread). At scale factor 1 the performance onsets Pon_i' are identical to their original Pon_i . At scale factor 0 all Pon_i' occur synchronously at the minimum of their originals (i.e. removed chord spread). At factor .5 a diminished chord spread will result, and at factor 2 an exaggerated chord spread can be obtained. At negative values of this factor the spread is inverted: first notes becoming last and vice versa. At extreme values of the scale factor the notes are restrained from moving out of the chord structure into the next musical object by truncation. Note that the whole operation is independent of score times.

Scaling the expressive asynchrony of an ACCIA section

The scaling of the expressive asynchrony of a collateral simultaneous structure works as follows. Assume the structure has a main component with performance onset time Pon_m and an ornament component with performance onset time Pon_o . A time-map of the structure contains Pon_o and Pon_m . The scale operation on such a map delivers new performance onsets according to the following rules:

Let the performance onset asynchrony ΔPon be defined as:

$$\Delta Pon = Pon_o - Pon_m$$

The asynchrony is scaled by a multiplication factor f , and a new performance onset Pon_o' is calculated:

$$\Delta Pon' = \Delta Pon * f$$

$$Pon_o' = Pon_m + \Delta Pon'$$

$$Pon_m' = Pon_m$$

Frame 9. Scaling the expressive asynchrony of an ACCIA section.

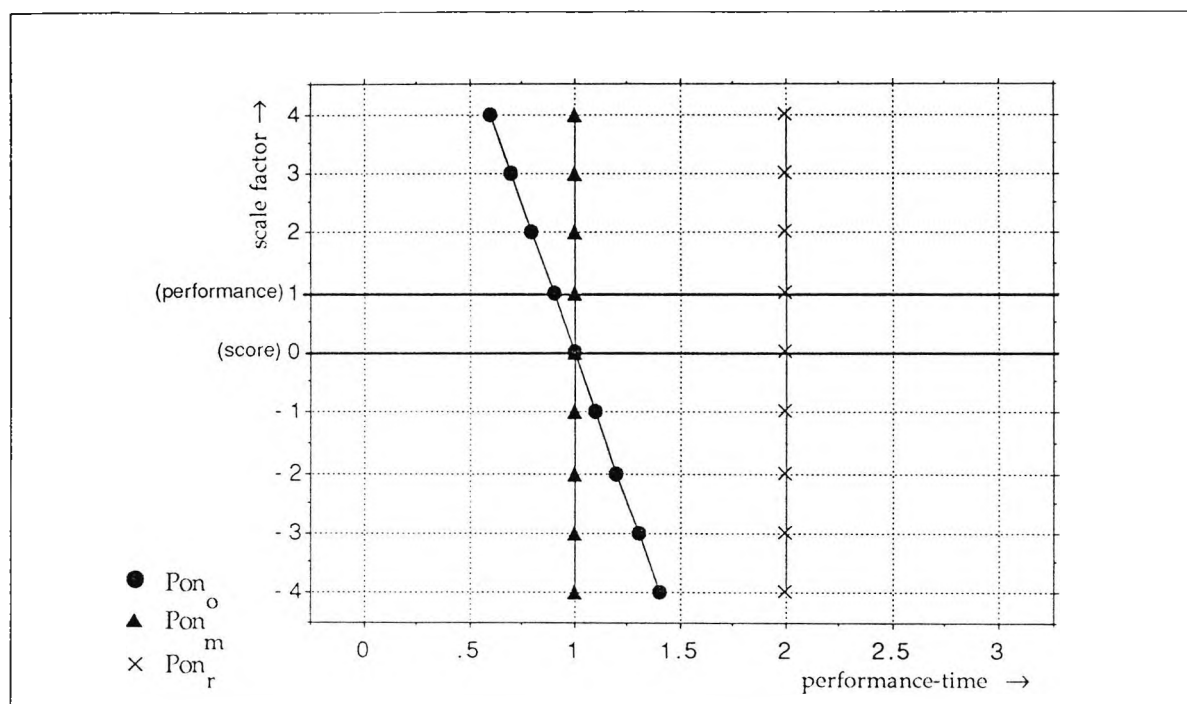


Figure 9. Scaling expressive timing of a ACCIA section. It shows this process for a specific set of performance times (a note preceded by an acciaccatura). At scale factor 1 all performance onsets are identical to their original. At scale factor 0 the ornament occurs synchronously with the main note (removed asynchrony). At negative values of this factor the order of onset of ornament and main note is inverted. Note that the ornament is allowed to shift freely - even outside the bounds of the whole ACCIA structure.

Scaling expressive articulation

The articulation of a note is interpreted (scaled) relative to the articulation of the structure that it forms part of. For multilateral structures this is the average articulation. If thus the first note in a bar is played with more overlap than the other notes, a removal of the overlap articulation expression (a zero scale factor) will set the overlap of all notes to the mean overlap of the notes in the structure. And exaggerating the articulation expression (a scale factor larger than 1) will move the individual overlaps away from the mean - but maintaining the average overlap of all the notes in the bar. Of course all articulation types maintain reasonable performance offsets in the case of extreme values (i.e. note offsets will not shift before their onsets).

Scaling the expressive articulation of a multilateral section

Assume a multilateral structure has n components C_i with $0 \leq i \leq n-1$. Component C_i has articulation A_i (see frame 5 for the calculation of A_i). A section of the expression map of the structure contains all A_i . The articulation A of the structure itself is defined as:

$$A = \text{MEAN}_{0 \leq i \leq n-1} A_i$$

Let the expression deviations be

$$\Delta A_i = A_i - A \text{ for } 0 \leq i \leq n-1$$

The deviations are simply scaled by a multiplication factor f

$$\Delta A_i' = f * \Delta A_i$$

The scale transformation delivers new articulations A_i' by adding the new deviations to the reference articulation A such that the articulation of the whole structure is kept invariant ($\text{mean } A_i' = A$).

$$A_i' = A + \Delta A_i'$$

Keeping the expressive values in a reasonable range can only be done while applying them to the individual notes.

Frame 10. Scaling the expressive articulation of a multilateral section

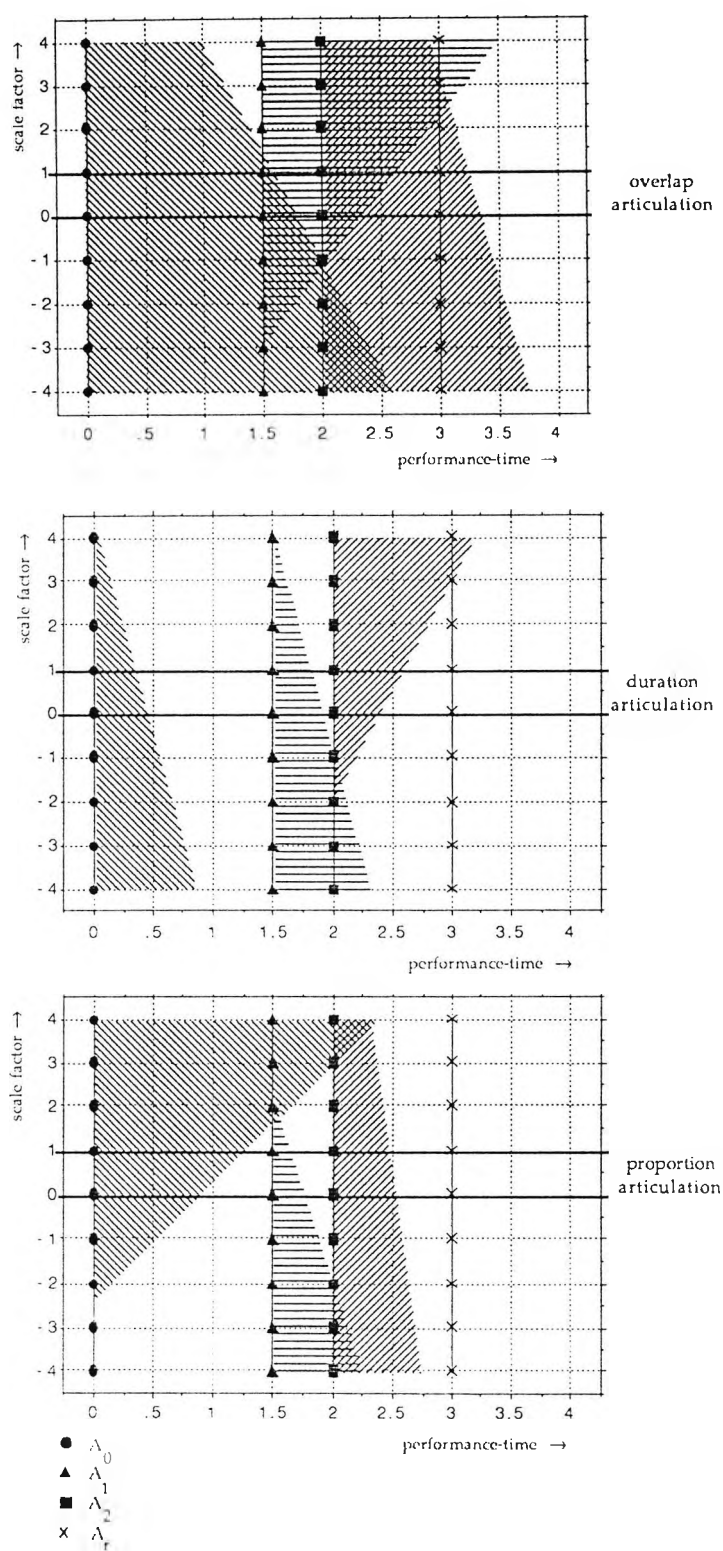


Figure 10. Scaling of an S section with three different kinds of articulation. It shows the scaling of three types of articulation for a multilateral structure, in this instance an S structure with a specific set of performance onset and offset times. Here, at scale factor 1 articulations A_i are identical to the original performance. At scale factor 0 all A_i are scaled to the mean articulation A . At a scale factor above 1 the deviation of each A_i with respect to A is exaggerated, with negative values constituting an inverse deviation: legato notes become more staccato and vice versa. Note that the mean articulation A is always kept invariant.

Scaling the expressive articulation of a collateral section

Assume that a collateral structure has ornament and main components C_o and C_m . Component C_o has articulation A_o and component C_m has articulation A_m (see frame 5 for the calculation of A_o and A_m). A section of the expression map of the structure contains these values. The articulation A of the structure itself is defined as:

$$A = A_m$$

Let the expression deviation be

$$\Delta A = A_o - A$$

The deviation is scaled by a multiplication factor f

$$\Delta A' = f * \Delta A$$

The scale transformation delivers a new articulation for the ornament by adding the new deviation to the reference articulation A .

$$A_o' = A + \Delta A'$$

$$A_m' = A_m$$

Keeping the expressive values in a reasonable range can only be done while applying them to the individual notes.

Frame 11. Scaling the expressive articulation of a collateral section.

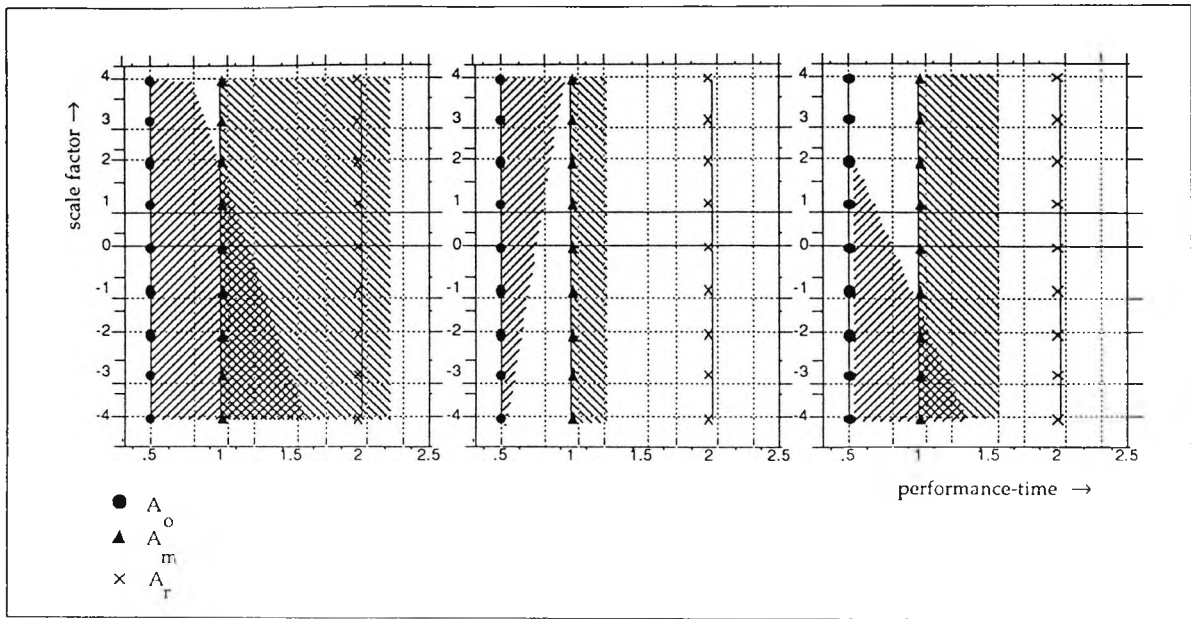


Figure 11. Scaling of an APPOG section with three different kinds of articulation. It shows three types of articulation scaling for an ornament (here an APPOG structure). At scale factor 1 the articulation A_o' is identical to the original articulation of the ornament. At scale factor 0 A_o is identical to the articulation of the main component A_m . At a scale factor above 1 the deviation of A_o with respect to the main component A_m is exaggerated, negative values constituting an inverse articulation: legato ornament articulation become more staccato and vice versa.

Keeping articulation consistent in the scaling of expressive timing

In the scaling of timing of onsets we ignored the influence it should have on its offsets. To obtain some sort of articulation consistency we can use the three types of articulation (as described above) when scaling expressive tempo and expressive asynchrony. In figure 12, we use expressive tempo scaling for an S section as an example in illustrating the different types of articulation consistency.

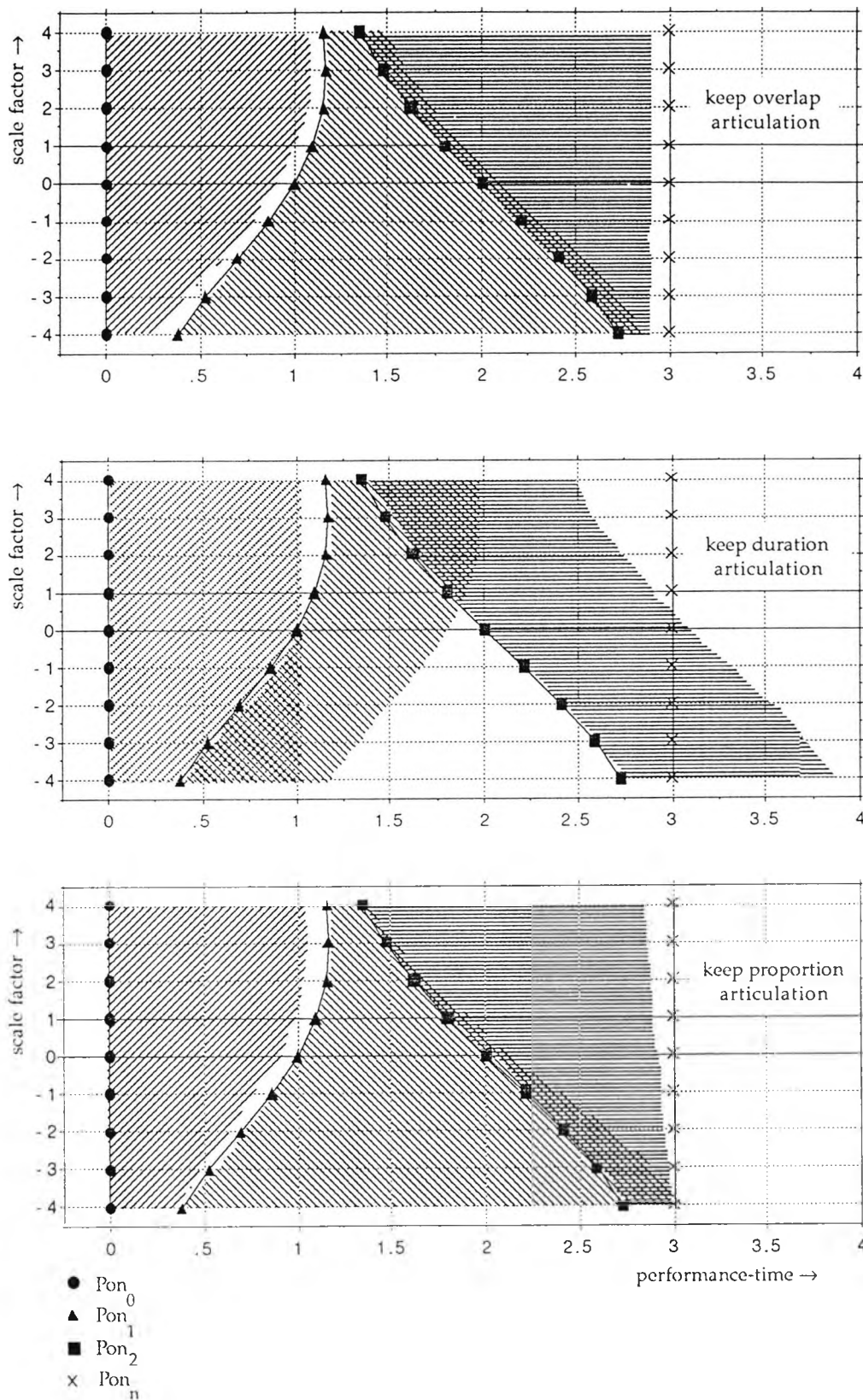


Figure 12. Scaling of an S section that keeps a particular type of articulation consistent. Shown for the same set of performance onsets as used in figure 6.

Stretch maps

Sometimes it is useful to be able to keep a consistency in performance timing between voices when modifying one of them. Naming the modified material as the foreground and describing the rest as the background, the consistency requires that a series of performance onsets, at a selected background level, that happen between two performance onsets in the foreground are “stretched along” with the changes in the foreground. This feature is implemented by first extracting a timing map from the background, and “stretching” this map between the old and modified foreground map before it is reapplied to the background. The fore- and background must be parallel (must happen during the same score time interval) and have to be S structures. Maintaining the consistency between other kinds of structure remains a problem.

Interpolate maps

A more sophisticated notion of expression entails the difference in expression between two structured objects. The best known example is voice leading in ensemble playing (Rasch, 1979) whereby the leading instrument often takes a small but consistent timing lead (around 10 ms). Inter- or extrapolation between two extracted timing maps yields the possibility to scale this kind of expression.

Transfer maps

Sometimes it is useful to apply an expression map extracted from one object, to another object, possibly with a different structure, e.g. boldly applying the expressive timing of the melody to the accompaniment. This is supported via an operation on timing maps that uses the structure of one map but imposes expressive values of the other.

TRANSFORMATIONS

Transformations of musical structures are generalizations of the operations on expression maps. They handle the selection of a level of structural description, extract a map, do the operation and re-impose the map. However, they often become quite sophisticated because they also take care of maintaining consistency with a background (material that is not affected directly). The application of the modified map has its own complexity, whereby changes are propagated to lower levels depending on the types of musical structure encountered. Finally, in the setting of new performance onsets of the notes, also the offsets may change in order to keep the articulation invariant. Out of the wealth of possibilities we choose some examples to be illustrated further by means of figures. In the figures the performance onsets and/or offsets of the individual notes at different parameter settings are given. The structure of the musical objects transformed are shown underneath.

In the following examples the same performance of a Beethoven theme is used (the fragment as shown in figure 2), allowing for comparison of the different transformations and to see the effect of applying the same transformation to different levels or types of structure. Note that for all the transformations the identity transformation is shown at scale factor 1.

Scale timing

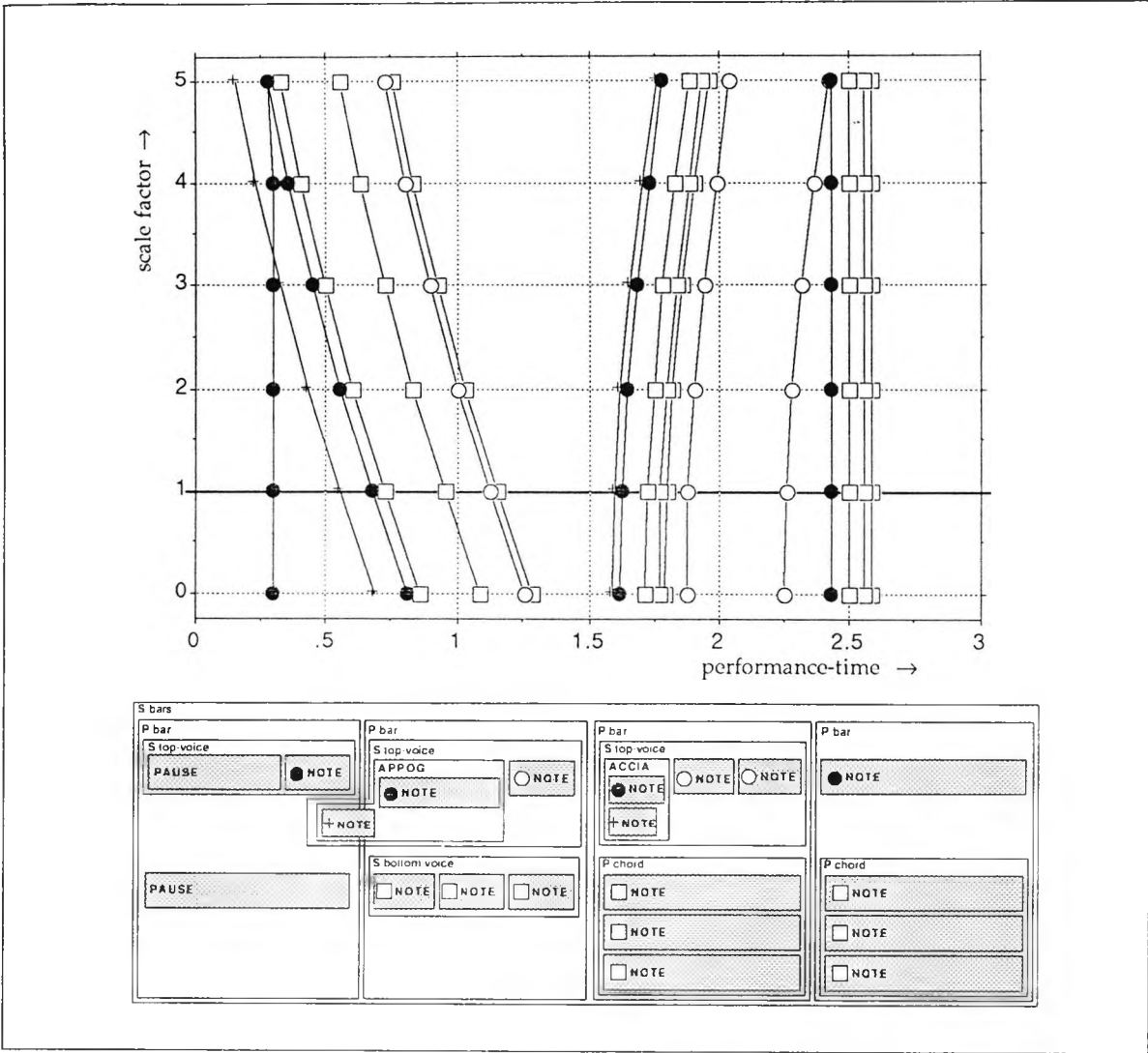


Figure 13. Scaling the expressive tempo of bars in the Beethoven fragment. Underneath the figure a structural description of the fragment is shown in bars. Imagine what would happen if we asked a performer to emphasize his/her timing of the bars? One possibility would be to play the onsets of the bars, that were played slightly early, even earlier, and ones that were played late, later still. This particular transformation can be read from figure 12 as the lines with the black markers, indicating the component in the bar that carries the expressive timing. Both the performance onset of the first and the last bar of the enclosing bars' structure are not changed; the transformation is done at the level named "bars", with its timing kept invariant. The lines with white markers show the embedded material that follows the change of the performance onset of each bar. Note that the timing of the ornamented notes does not change (they keep the same distance with respect to the note they cling to), as does the spread of the chords.

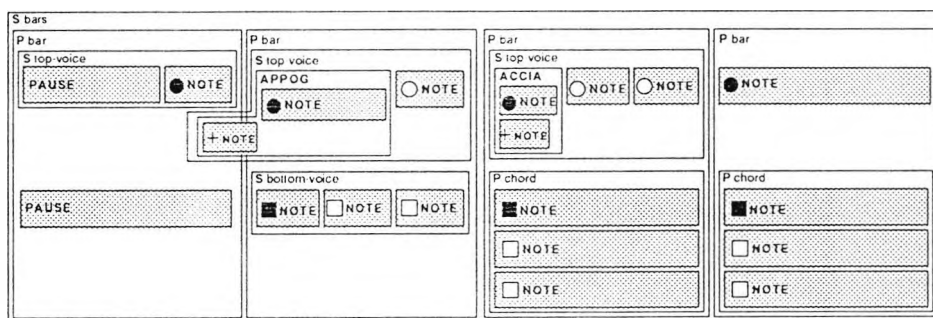
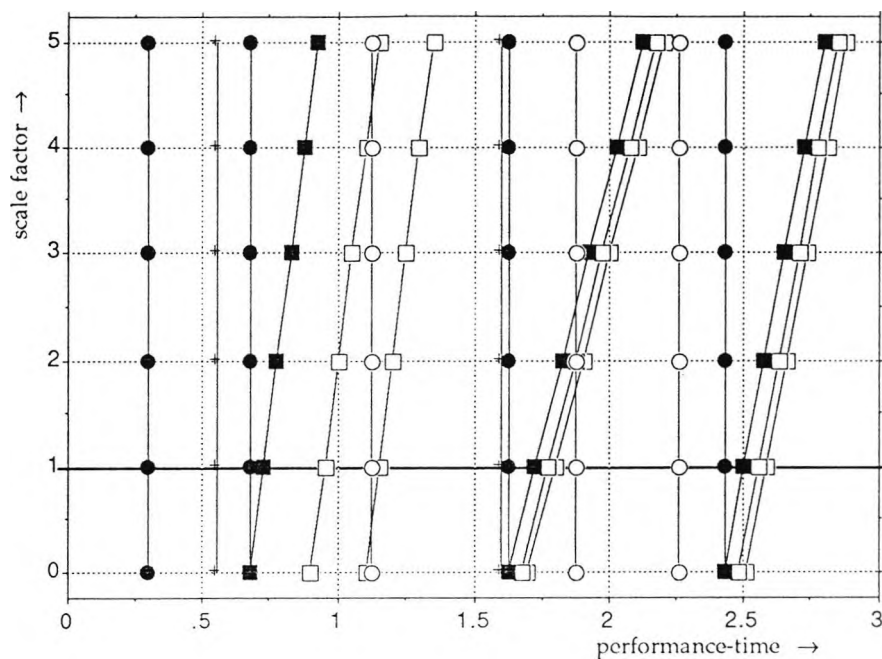


Figure 14. Scaling the expressive asynchrony of each bar in the Beethoven fragment. It shows the expressive transformation we might expect to happen when a performer is asked to exaggerate the asynchrony between the top-voice and bottom-voice at the onset of each bar. The figure shows the scaling of the asynchrony of the bottom voice onsets (the black squares), without changing the timing of the bars (lines marked with black circles and triangles). The embedded notes of the bottom voice (lines with white squares) just shift along with the expressive timing of their embedding structure. Here again, the ornament timing and the chord spread stay invariant.

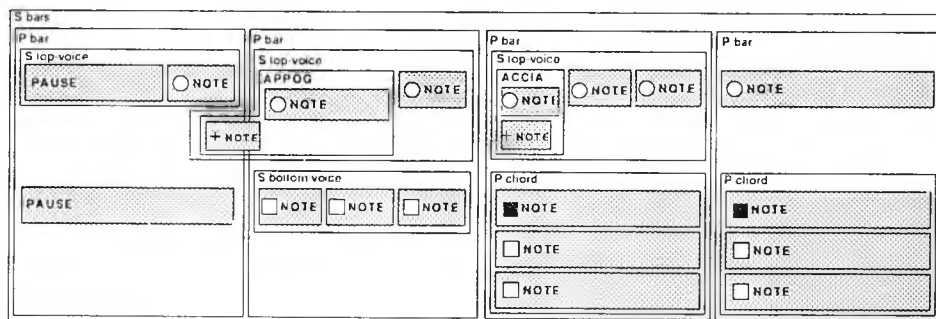
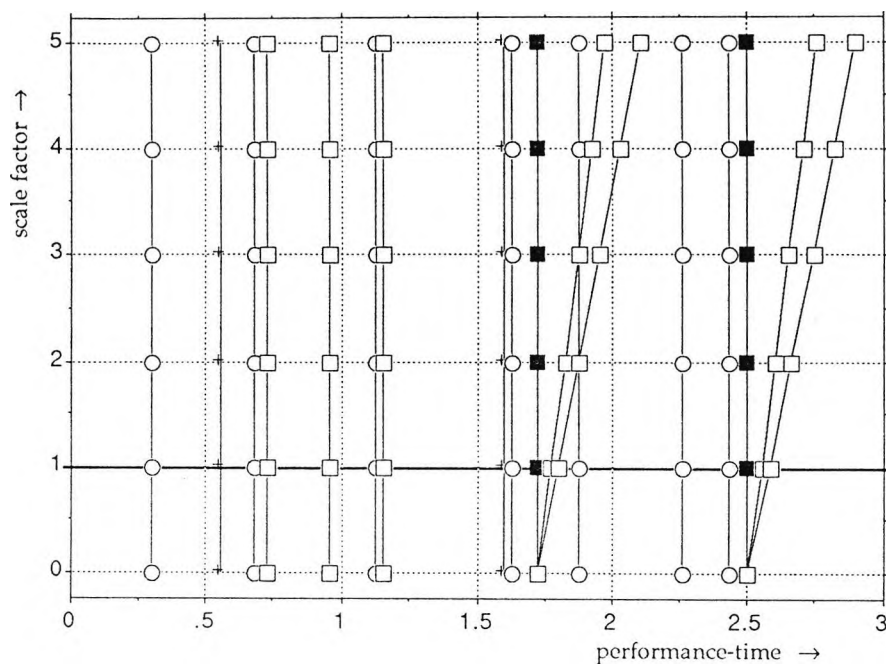


Figure 15. Scaling the expressive asynchrony of each chord in the Beethoven fragment. It shows another expressive transformation that exaggerates the chord spread, turning them almost into arpeggio's at high scale factors. At scale factor 0 the chord spread is completely removed. The timing of the rest of the fragment stays unaltered.

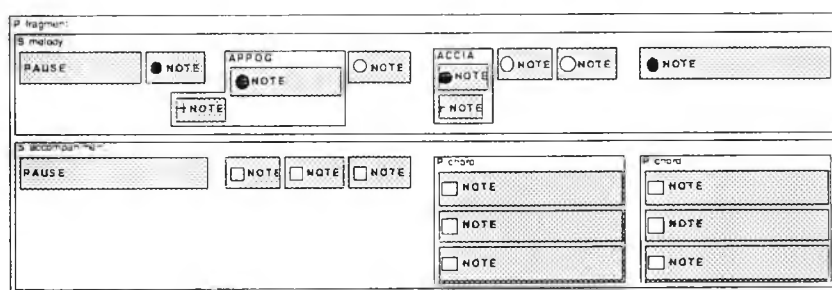
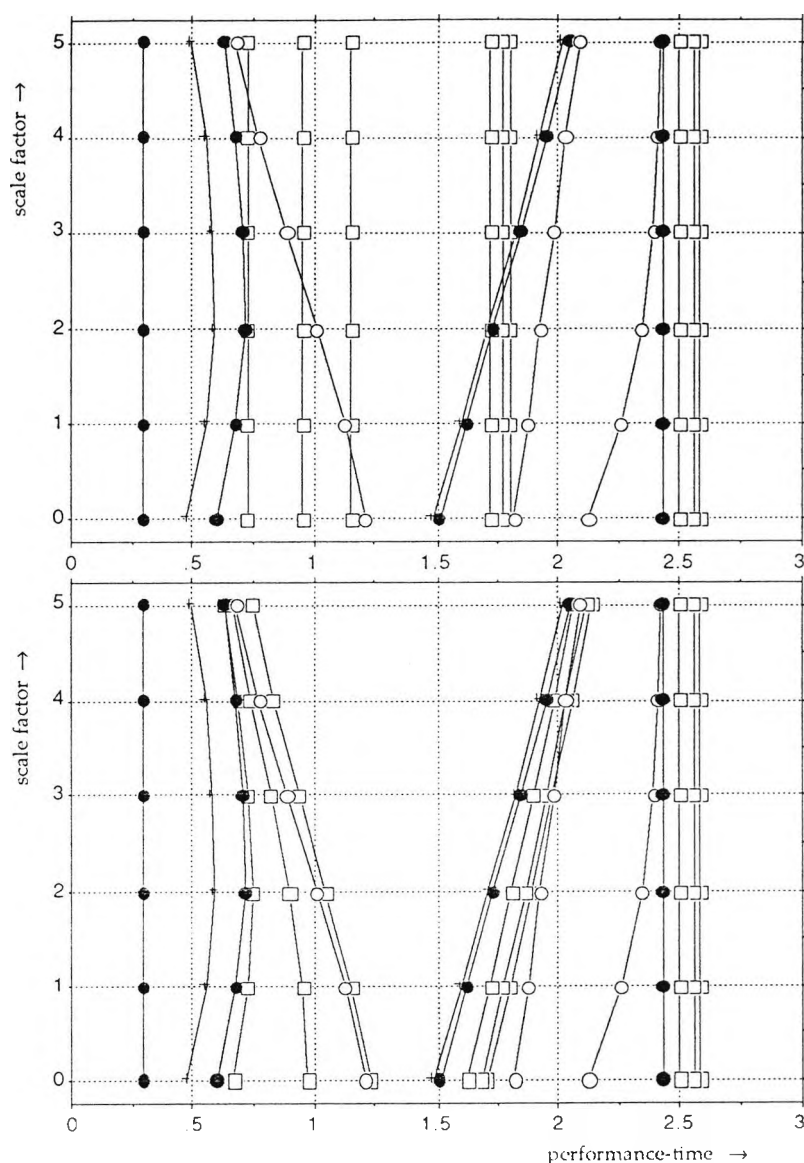


Figure 16. Scaling the expressive tempo of the melody in the Beethoven fragment, a) without and b) with “stretching” the accompaniment. It shows that the timing of each note of the melody becomes exaggerated with a higher scale factor. Here the accompaniment (lines marked with white squares) is not affected at all. Figure 16b, on the other hand, shows a musically more reasonable transformation: the accompaniment follows the movements of the transformed melody, e.g. slowing down when the tempo of the melody slows down. Here the accompaniment is kept consistent with respect to the original performance (compare with the onsets at scale factor 1). Note that note order can change between melody and accompaniment, because of the structural description in two parallel voices.

Keeping articulation consistent

In the above examples we showed the scaling of onset times and neglected what happened to the offset times. But, as we showed before, this cannot just be ignored in musically relevant transformations. We can select one of the described types of articulation to keep consistent, but we do not show this here (see figure 12 for a simple example).

Scale intervoice expression

When the expression between voices is scaled, two parameters are used. The first one selects a reference level of expression (0 designates the expression of the first, 1 designates the expression of the second, 0.5 is the mean of the two etc.). The second parameter determines in how far the voices are removed from that reference level (0 means completely on reference level, 1 means as in original performance, 2 means exaggerated with respect to the reference etc.).

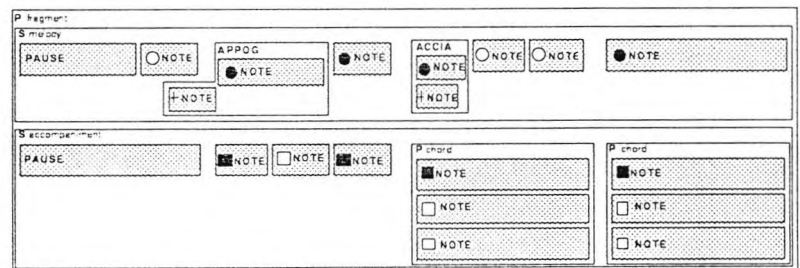
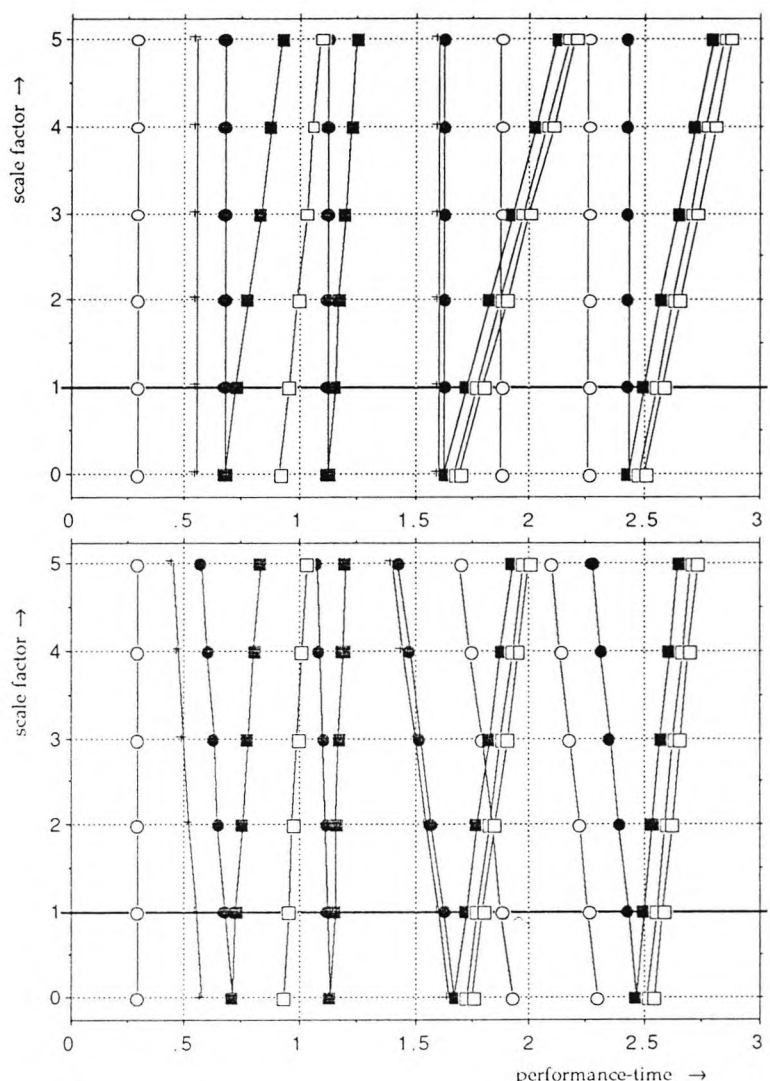


Figure 17. Scaling the intervoice timing between the melody and the accompaniment in the Beethoven fragment, a) with the melody as reference, and b) with the mean of the melody and the accompaniment as reference. In figure 17a intervoice timing (one type of intervoice scaling) is scaled with the melody voice as the reference. It shows the scaling of the asynchrony between the accompaniment and the melody, as found in the performance (see the horizontal line where the scale factor is 1). Notes that are not synchronous (i.e. don't have the same score time) interpolate their change with respect to their surrounding performance onsets that are considered parallel (have the same score time). Note that the timing of the melody does not change because it is used as reference. In figure 17b the mean of the melody and accompaniment timing is used as reference, resulting in displacements (with respect to this invisible reference) of both voices. In both figures, the first event in the melody voice is unaffected since there is no measurable timing in the accompaniment (only a rest).

CONCLUSION

In this paper we have presented a proposal for a calculus that enables expressive timing to be transformed on the basis of structural aspects. The program implementing the calculus, will hopefully prove to be a solid basis for formalised theories of music cognition. A micro version of this program is included in the appendix, open to further inquiry and immediate test. The proposed representation constructs allow for easy maintenance and extension. An object-oriented programming style proved a good choice for this kind of modelling. The algorithmic parts became reasonably simple, but the program can still be considered as quite complex, especially its elaborate knowledge representation. This algorithmic simplicity combined with structural complexity mirrors, in this respect, the widespread hypothesis that the complex expressive timing profiles found in musical performances are more readily explained as the product of a small collection of simple rules linked to a relatively complex structure, than as the result of a large collection of interacting rules, with hardly any structure.

This research again confirmed that music is a very rewarding field for experimentation with knowledge representation concepts.

ACKNOWLEDGEMENTS

We are very grateful to Eric Clarke who made it possible for us to work for two years on research in expressive timing at City University in London, and the British ESRC for their financial support (grant A413254004) during this period.

REFERENCES

- Bentley J. (1988) More Programming Pearls, Confessions of a Coder. Reading, MA: Addison-Wesley.
- Bregman, A. S. (1990) Auditory Scene Analysis: The Perceptual Organization of Sound. Cambridge, Mass.: Bradford books, MIT Press.
- Clarke, E. F. (1988) Generative principles in music performance. In J. A. Sloboda (Ed.) Generative processes in music. The psychology of performance, improvisation and composition. Oxford: Science Publications.
- Desain, P. & H. Honing (1988) LOCO: A Composition Microworld in Logo. Computer Music Journal 12(3): 30-42.
- Desain, P. & H. Honing (1991) Quantization of Musical Time: A Connectionist Approach. In P. M. Todd & G. J. Loy (Eds.) Music and Connectionism. Cambridge, Mass.: MIT Press.
- Desain, P. & H. Honing (in press, a) Tempo curves considered harmful. To appear in Contemporary Music Review.

- Desain, P. & H. Honing (in press, b) Time functions function best as functions of multiple times.
To appear in Computer Music Journal.
- Desain, P. (1990) Lisp as a second language. Perspectives of New Music 28(1).
- Honing, H. (1990) POCO: An Environment for Analysing, Modifying, and Generating Expression in Music. In Proceedings of the 1990 International Computer Music Conference.
San Francisco: Computer Music Association.
- Honing, H. (1991) Issues in the Representation of Time and Structure in Music. In Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press. (forthcoming).
- Keene, S. E. (1989) Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. Reading, MA: Addison-Wesley.
- Longuet-Higgins, H. C. (1976) The Perception of Melodies. Nature 263: 646-653.
- Rasch, R. A. (1979) Synchronisation in performed ensemble music. Acoustica 43, 121-131.
- Serafine, M.L. (1988) Music as Cognition: The Development of Thought in Sound. New York: Columbia University Press.
- Steele, G. L. (1990) Common Lisp, the Language. Second edition. Bedford, MA: Digital Press.
- Vos, J. & R. A. Rasch (1981) The perceptual onset of musical tones. Perception & Psychophysics 29(4): 323-335.

MICROWORLD EXPRESSION CALCULUS

```
;*****
; A CALCULUS FOR MUSIC PERFORMANCE EXPRESSION
; (c) 1991, Peter Desain & Henkjan Honing
;
; in CLOS (Common Lisp), uses loop macro
;*****

;*****
; MUSICAL OBJECTS
;*****
; abstract classes of musical objects

(defclass musical-object ()
  ((name :reader name :initarg :name :initform 'no-name :type symbol)
   (score-onset :reader score-onset :type rational :initform 0)
   (left :reader left :initform nil)
   (right :reader right :initform nil))
  (:documentation "Musical Object"))

(defclass structured (musical-object)
  ((score-offset :reader score-offset :type rational))
  (:documentation "Structured Musical Object"))

(defclass multilateral (structured)
  ((components :reader components :initarg :components))
  (:documentation "Multilateral Musical Object"))

(defclass collateral (structured)
  ((main :reader main :initarg :main)
   (ornament :reader ornament :initarg :ornament))
  (:documentation "Ornamented Musical Object"))

(defclass successive (structured)
  ()
  (:documentation "Successive Musical Object"))

(defclass simultaneous (structured)
  ()
  (:documentation "Simultaneous Musical Object"))

(defclass basic (musical-object)
  ((score-offset :reader score-offset :type rational :initarg :score-dur))
  (:documentation "Basic Musical Object"))

;*****
; instantiatable classes of musical objects

(defclass S (multilateral successive) () (:documentation "Sequential"))
(defclass P (multilateral simultaneous) () (:documentation "Parallel"))
(defclass ACCIA (collateral simultaneous) () (:documentation "Acciaccature"))
(defclass APPOG (collateral successive) () (:documentation "Appoggiature"))

(defclass NOTE (basic)
  ((dynamic :accessor dynamic :type float :initarg :dynamic)
   (perf-onset :accessor perf-onset :type float :initarg :perf-onset :initform nil)
   (perf-offset :accessor perf-offset :type float :initarg :perf-offset :initform nil))
  (:documentation "Note"))

(defclass PAUSE (basic) () (:documentation "Rest"))
```

```

;*****
; creators for musical objects

(defun S (name &rest components)
  (make-instance 'S :name name :components components))

(defun P (name &rest components)
  (make-instance 'P :name name :components components))

(defun ACCIA (name ornament main)
  (make-instance 'ACCIA :name name :ornament ornament :main main))

(defun APPOG (name ornament main)
  (make-instance 'APPOG :name name :ornament ornament :main main))

(defun NOTE (&key name perf-onset perf-offset score-dur (dynamic 1))
  (make-instance 'NOTE :name name
    :perf-onset perf-onset
    :perf-offset perf-offset
    :score-dur score-dur
    :dynamic dynamic))

(defun PAUSE (&key name score-dur)
  (make-instance 'PAUSE :name name :score-dur score-dur))

;*****
; extra access functions for musical objects

(defmethod components ((object basic)) nil)
(defmethod components ((object collateral))
  (list (ornament object) (main object)))

(defmethod all-notes ((object musical-object))
  (loop for component in (components object) append (all-notes component)))

(defmethod all-notes ((object note)) (list object))

(defun has-name? (&rest names)
  #'(lambda (object &rest ignore) (member (name object) names)))

(defmethod find-parts ((object musical-object) pred)
  (if (funcall pred object)
    (list object)
    (loop for component in (components object)
      append (find-parts component pred))))

;*****
; initialization of score times and context of musical objects

(defmethod initialize-instance :after ((object musical-object) &rest ignore)
  (object-check object)
  (initialize-score-times object)
  (initialize-context object))

(defmethod object-check ((object musical-object)) nil)

;*****
; initialization of score-onset and offset of musical objects

(defmethod initialize-score-times ((object basic))

(defmethod initialize-score-times ((object P))
  (setf (slot-value object 'score-offset)
    (slot-value (first (components object)) 'score-offset)))

```

```

(defmethod initialize-score-times ((object S))
  (loop with onset = 0
    for component in (components object)
    do (shift-score component onset)
    (setf onset (slot-value component 'score-offset))
    finally (setf (slot-value object 'score-offset) onset)))

(defmethod initialize-score-times ((object collateral))
  (setf (slot-value object 'score-offset)
    (slot-value (main object) 'score-offset)))

(defmethod initialize-score-times :after ((object APPOG))
  (shift-score (ornament object)
    (- (slot-value (ornament object) 'score-offset))))

(defmethod shift-score ((object musical-object) shift)
  (incf (slot-value object 'score-onset) shift)
  (incf (slot-value object 'score-offset) shift)
  (loop for component in (components object) do (shift-score component shift)))

;*****
; initialization of context of musical objects

(defmethod initialize-context ((object musical-object))

(defmethod initialize-context ((object S))
  (loop for component in (components object)
    for next-component in (rest (components object))
    do (set-contexts component next-component)))

(defmethod initialize-context ((object APPOG))
  (set-context (ornament object) (main object) 'right))

(defmethod set-contexts ((left musical-object) (right musical-object))
  (set-context left right 'right)
  (set-context right left 'left))

(defmethod set-context ((object musical-object) (context musical-object) dir)
  (setf (slot-value object dir) context))

(defmethod set-context :after ((object P) (context musical-object) dir)
  (loop for component in (components object)
    do (set-context component context dir)))

(defmethod set-context :after ((object S) (context musical-object) dir)
  (if (eq dir 'left)
    (set-context (first (components object)) context dir)
    (set-context (last-element (components object)) context dir)))

(defmethod set-context :after ((object collateral) (context musical-object) dir)
  (set-context (main object) context dir))

(defmethod set-context :after ((object ACCIA) (context musical-object) dir)
  (when (eq dir 'left)
    (set-context (ornament object) context dir)))

```

```

;*****
;*****
; MAPS
;*****
;*****
; abstract classes of maps

(defclass map ()
  ((sections :accessor sections :initarg :sections)
   (:documentation "Expression Map"))

(defclass multilateral-map (map) ())
(defclass collateral-map (map) ())
(defclass simultaneous-map (map) ())
(defclass successive-map (map) ())

;*****
; instantiable classes of maps

(defclass P-map (multilateral-map simultaneous-map) ())
(defclass S-map (multilateral-map successive-map) ())
(defclass ACCIA-map (collateral-map simultaneous-map) ())
(defclass APPOG-map (collateral-map successive-map) ())

;*****
; creator for maps

(defun make-map (sections)
  (let ((ordered-sections (sort sections #'< :key #'score-onset)))
    (cond ((null ordered-sections) nil)
          ((and (same-section-type? ordered-sections)
                (not-overlapping? ordered-sections))
           (make-instance (section-to-map (first ordered-sections))
                          :sections ordered-sections))
          (t (error "attempt to merge incompatible sections into expression map")))))

;*****
; sections of maps
;*****
; abstract classes of sections of maps

(defclass section ()
  ((all-score-times :accessor all-score-times :initarg :all-score-times)
   (all-expressions :accessor all-expressions :initarg :all-expressions)
   (:documentation "Expression Section"))

(defclass multilateral-section (section) ())
(defclass collateral-section (section) ())
(defclass successive-section (section) ())
(defclass simultaneous-section (section) ())

;*****
; instantiable classes of sections of maps

(defclass S-section (successive-section multilateral-section) ())
(defclass P-section (simultaneous-section multilateral-section) ())
(defclass ACCIA-section (simultaneous-section collateral-section) ())
(defclass APPOG-section (successive-section collateral-section) ())

;*****
; compatibility relation between musical objects, expression maps and sections thereof

(defmethod object-to-section ((object musical-object))
  (third (find (class-name (class-of object)) (object-network) :key #'first)))

(defmethod section-to-map ((section section))
  (second (find (class-name (class-of section)) (object-network) :key #'third)))

```

```

(defun object-network ()
  '((S S-map S-section)
    (P P-map P-section)
    (ACCIA ACCIA-map ACCIA-section)
    (APPOG APPOG-map APPOG-section)))

;*****
; creators for sections of maps

(defun make-section (section-class all-score-times all-expressions)
  (make-instance section-class
    :all-score-times all-score-times
    :all-expressions all-expressions))

(defmethod make-new-section ((section section) expressions)
  (make-section (class-of section)
    (snoc (score-times section) (score-offset section))
    (snoc expressions (next-expression section))))

(defmethod make-new-section-from-pairs ((section section) pairs)
  (make-section (class-of section)
    (snoc (mapcar #'first pairs) (score-offset section))
    (snoc (mapcar #'second pairs) (next-expression section))))

;*****
; extra accessors for sections of maps

(defmethod score-onset ((section section))
  (first (all-score-times section)))

(defmethod score-offset ((section section))
  (last-element (all-score-times section)))

(defmethod expressions ((section section))
  (butlast (all-expressions section)))

(defmethod next-expression ((section section))
  (last-element (all-expressions section)))

(defmethod score-times ((section section))
  (butlast (all-score-times section)))

(defmethod score-onset ((section collateral-section))
  (score-main section))

(defmethod main-expression ((section collateral-section))
  (second (all-expressions section)))

(defmethod ornament-expression ((section collateral-section))
  (first (all-expressions section)))

(defmethod score-main ((section collateral-section))
  (second (all-score-times section)))

(defmethod score-ornament ((section collateral-section))
  (first (all-score-times section)))

(defun same-section-type? (sections)
  (every #'(lambda (section) (class-of section)) sections))

(defun not-overlapping? (sections)
  (loop for section in sections
    for next-section in (rest sections)
    never (> (score-offset section) (score-onset next-section))))

```

```

;*****
; find section (containing score time) in expression map

(defmethod lookup-section-containing ((map map) score-time)
  (loop for section in (sections map)
    when (<= (score-onset section) score-time (score-offset section))
    do (return section)))

;*****
; lookup expression value (via score time) in expression map

(defmethod lookup-defined-expression ((map map) score-time)
  (lookup-defined-expression (lookup-section-containing map score-time) score-time))

(defmethod lookup-defined-expression (section score-time)
  (and section
    (loop for expression in (all-expressions section)
      for map-score-time in (all-score-times section)
      when (= map-score-time score-time)
      do (return expression))))

(defmethod lookup-expression ((map successive-map) score-time)
  (lookup-expression (lookup-section-containing map score-time) score-time))

(defmethod lookup-expression (section score)
  (and section
    (loop for expression in (all-expressions section)
      for expression-next in (rest (all-expressions section))
      for score-time in (all-score-times section)
      for score-time-next in (rest (all-score-times section))
      while (> score score-time-next)
      finally (return (interpolate score-time score score-time-next
        expression expression-next))))))

;*****
; lookup score time in a monotone rising expression map

(defmethod in-section-inverse? ((section section) expression)
  (and expression (<= (first (expressions section))
    expression
    (or (next-expression section)
      (last-element (expressions section))))))

(defmethod lookup-inverse ((map S-map) expression)
  (loop for section in (sections map) thereis (lookup-inverse section expression)))

(defmethod lookup-inverse ((section section) expression)
  (and (in-section-inverse? section expression)
    (loop for expression-next in (rest (expressions section))
      for score-time in (score-times section)
      for score-time-next in (rest (score-times section))
      while (> expression expression-next)
      finally (return (list score-time score-time-next)))))

;*****
; mapping through expression maps

(defmethod map-map (fun (map map))
  (make-map (loop for section in (sections map) collect (funcall fun section))))

;*****
; mapping through filtered expression maps

(defmethod with-filtered-null-expression (fun (map map))
  (unfilter-null-expression (funcall fun (filter-null-expression map))
    (filter-null-expression-out map)))

(defmethod filter-null-expression ((map map))
  (map-map #'filter-null-expression map))

```

```

(defmethod filter-null-expression ((section section))
  (make-new-section-from-pairs section
    (loop for expression in (expressions section)
      for score-time in (score-times section)
      when expression
      collect (list score-time expression))))

(defmethod filter-null-expression-out ((map map))
  (mapcar #'filter-null-expression-out (sections map)))

(defmethod filter-null-expression-out ((section section))
  (loop for expression in (expressions section)
    for score-time in (score-times section)
    for index from 0
    unless expression
    collect (list index score-time)))

(defmethod unfilter-null-expression ((map map) rejections)
  (make-map (mapcar #'unfilter-null-expression (sections map) rejections)))

(defmethod unfilter-null-expression ((section section) removed)
  (if removed
    (make-new-section-from-pairs section
      (loop with expressions = (expressions section)
        with score-times = (score-times section)
        for index from 0
        while (or score-times removed)
        when (and removed (= index (caar removed)))
        collect (list (second (pop removed)) nil)
        else collect (list (pop score-times)
          (pop expressions))))
    section))

```

```

;*****
;*****
; EXPRESSION
;*****
;*****

(defclass expression () ())

;*****
; nil and rests carry no expression, nil expressions and sections are not set

(defmethod get-expression ((object null) (expression expression)) nil)
(defmethod get-next-expression ((object null) (expression expression)) nil)

(defmethod get-expression ((object PAUSE) (expression expression)) nil)
(defmethod set-expression ((object PAUSE) (expression expression) value) nil)

(defmethod set-expression ((object musical-object) expression value-or-section) nil)
(defmethod get-next-expression ((object musical-object) (expression expression))
  (get-expression (right object) expression))

;*****
; get expression of notes

(defmethod get-notes-expression ((object musical-object) (expression expression))
  (loop for note in (all-notes object)
    collect (fetch-expression note expression)))

(defmethod set-notes-expression ((object musical-object) (expression expression) values)
  (loop for note in (all-notes object)
    for value in values
    do (set-expression note expression value)))

;*****
; propagate expression (interpolated, truncating-shift and shift)

(defmethod propagate-interpolated ((object S)
  old-begin new-begin old-end new-end expression)
  (loop for component in (components object)
    do (propagate-interpolated component
      old-begin new-begin old-end new-end expression)))

(defmethod propagate-interpolated ((object P)
  old-begin new-begin old-end new-end expression)
  (loop for component in (components object)
    do (propagate-truncating-shift component
      (save-- new-begin old-begin) new-end expression)))

(defmethod propagate-interpolated ((object collateral)
  old-begin new-begin old-end new-end expression)
  (let* ((ref (fetch-expression (main object) expression))
    (shift (save-- (interpolate old-begin ref old-end new-begin new-end) ref)))
    (propagate-interpolated (main object) old-begin new-begin old-end new-end expression)
    (propagate-shift (ornament object) shift expression)))

(defmethod propagate-interpolated ((object NOTE)
  old-begin new-begin old-end new-end expression)
  (set-expression
    object expression
    (interpolate old-begin (fetch-expression object expression)
      old-end new-begin new-end)))

(defmethod propagate-interpolated ((object PAUSE)
  old-begin new-begin old-end new-end expression))

```



```

;*****
; propagate-truncating-shift

(defmethod propagate-truncating-shift :around ((object musical-object)
                                              shift end expression)
  (when shift (call-next-method)))

(defmethod propagate-truncating-shift ((object multilateral) shift end expression)
  (loop for component in (components object)
        do (propagate-truncating-shift component shift end expression)))

(defmethod propagate-truncating-shift ((object collateral) shift end expression)
  (propagate-shift (ornament object) shift expression)
  (propagate-truncating-shift (main object) shift end expression))

(defmethod propagate-truncating-shift ((object NOTE) shift end expression)
  (set-expression object
                  expression
                  (save-min (save++ (fetch-expression object expression) shift) end)))

(defmethod propagate-truncating-shift ((object PAUSE) shift end expression))

;*****
; propagate-shift

(defmethod propagate-shift :around ((object musical-object) shift expression)
  (when shift (call-next-method)))

(defmethod propagate-shift ((object structured) shift expression)
  (loop for component in (components object)
        do (propagate-shift component shift expression)))

(defmethod propagate-shift ((object basic) shift expression)
  (set-expression object
                  expression
                  (save++ (fetch-expression object expression) shift)))

;*****
; onset timing
;*****

(defclass expressive-timing (expression) ())
(defclass onset-timing      (expressive-timing) ())
(defclass basic-asynchrony  (onset-timing) ())
(defclass basic-tempo       (onset-timing) ())

(defclass estimate-onset-timing (onset-timing estimate-mixin) ())

;*****
; get expressive timing

(defmethod get-expression ((object NOTE) (expression onset-timing))
  (perf-onset object))

(defmethod get-expression ((object S) (expression onset-timing))
  (get-expression (first (components object)) expression))

(defmethod get-expression ((object P) (expression onset-timing))
  (loop for component in (components object)
        when (get-expression component expression)
        minimize it))

(defmethod get-expression ((object collateral) (expression onset-timing))
  (get-expression (main object) expression))

```

```

;*****
; set expressive timing

(defmethod set-expression ((object NOTE) (expression onset-timing) value)
  (setf (perf-onset object) value))

(defmethod set-expression ((object S) (expression onset-timing) (section S-section))
  (loop for new-expression in (expressions section)
        for next-new-expression in (snoc (rest (expressions section))
                                          (next-expression section))
        for component in (components object)
        do (propagate-interpolated component
                                   (fetch-expression component expression)
                                   new-expression
                                   (fetch-expression (right component) expression)
                                   next-new-expression
                                   expression)))

(defmethod set-expression ((object P) (expression onset-timing) (section P-section))
  (loop for new-expression in (expressions section)
        for component in (components object)
        do (propagate-truncating-shift component
                                       (save-- new-expression
                                             (fetch-expression component expression))
                                       (get-next-expression object expression)
                                       expression)))

(defmethod set-expression ((object ACCIA)
                           (expression onset-timing)
                           (section ACCIA-section))
  (propagate-shift (ornament object)
                   (save-- (ornament-expression section)
                           (fetch-expression (ornament object) expression))
                   expression))

(defmethod set-expression ((object APPOG)
                           (expression onset-timing)
                           (section APPOG-section))
  (propagate-interpolated (ornament object)
                          (fetch-expression (ornament object) expression)
                          (ornament-expression section)
                          (fetch-expression (right (ornament object)) expression)
                          (main-expression section)
                          expression))

;*****
; scale expressive-timing

(defmethod scale-expression ((section P-section)
                            (expression basic-asynchrony)
                            factor)
  (if (expressions section)
      (make-new-section
       section
       (scale-P-expression-points (expressions section) factor))
      section))

(defmethod scale-expression ((section S-section)
                            (expression basic-tempo)
                            factor)
  (cond ((and (expressions section) (next-expression section))
        (scale-S-section-] section factor))
        ((rest (expressions section))
         (scale-S-section-> section factor))
        (t section)))

```

```

(defmethod scale-S-section-] ((section section) factor)
  (make-new-section section (scale-S-expression-points
    (snoc (score-times section) (score-offset section))
    (snoc (expressions section) (next-expression section))
    factor)))

(defmethod scale-S-section-> ((section section) factor)
  (make-new-section section
    (scale-S-expression-points (score-times section)
      (expressions section)
      factor)))

(defmethod scale-expression ((section ACCIA-section)
  (expression basic-asynchrony) factor)
  (make-new-section section
    (scale-ACCIA-points (main-expression section)
      (ornament-expression section)
      factor)))

(defmethod scale-expression ((section APPOG-section) (expression basic-tempo) factor)
  (make-new-section section
    (scale-APPOG-points (ornament-expression section)
      (main-expression section)
      (next-expression section)
      (score-ornament section)
      (score-main section)
      (score-offset section)
      factor)))

;*****

(defun scale-P-expression-points (perf-onsets factor)
  (let* ((perf-begin (apply #'min perf-onsets))
    (perf-iois (mapcar #'(lambda (onset) (- onset perf-begin)) perf-onsets))
    (raw-new-perf-iois (mapcar #'(lambda (perf) (scale-expression-lin perf factor))
      perf-iois))
    (shift (- (apply #'min raw-new-perf-iois)))
    (new-perf-onsets (mapcar #'(lambda (ioi) (+ ioi shift perf-begin))
      raw-new-perf-iois)))
    new-perf-onsets))

(defun scale-S-expression-points (score-times perf-times factor)
  (let* ((perf-iois (mapcar #'(- (rest perf-times) perf-times))
    (score-iois (mapcar #'(- (rest score-times) score-times))
    (perf-begin (first perf-times))
    (perf-end (last-element perf-times))
    (raw-new-perf-iois (mapcar #'(lambda (score perf)
      (scale-velocity score perf factor))
      score-iois
      perf-iois))
    (new-perf-iois (normalise raw-new-perf-iois (- perf-end perf-begin)))
    (new-perf-times (integrate new-perf-iois perf-begin)))
    new-perf-times))

(defun scale-ACCIA-points (main-expression ornament-expression factor)
  (let* ((expression-interval (- main-expression ornament-expression))
    (new-expression-ornament (- main-expression
      (scale-expression-lin expression-interval factor))))
    (list new-expression-ornament main-expression)))

```

```

(defun scale-APPOG-points (ornament-expression main-expression next-expression
                          score-ornament score-main score-end
                          factor)
  (let* ((score-ornament-ioi (- score-main score-ornament))
         (expression-ornament-ioi (- main-expression ornament-expression))
         (score-main-ioi (- score-end score-main))
         (expression-main-ioi (- next-expression main-expression))
         (ornament-tempo (/ score-ornament-ioi expression-ornament-ioi))
         (main-tempo (/ score-main-ioi expression-main-ioi))
         (relative-tempo (/ ornament-tempo main-tempo))
         (new-ornament-tempo (* main-tempo (expt relative-tempo factor)))
         (new-expression-ornament-ioi (/ score-ornament-ioi new-ornament-tempo))
         (new-expression-ornament (- main-expression new-expression-ornament-ioi)))
    (list new-expression-ornament main-expression next-expression)))

;*****
; expression scale methods

(defun scale-velocity (score perf factor)
  "Exponential scaling"
  (/ score (expt (/ score perf) factor)
    ))

(defun scale-expression-lin (perf factor)
  "Linear scaling"
  (* perf factor))

;*****
; stretch expressive-timing

(defmethod stretch-expression ((section S-section)
                               (old S-map)
                               (new S-map)
                               (expression onset-timing))
  (make-new-section
   section
   (loop for perf-time in (expressions section)
        as (score-begin score-end) = (lookup-inverse old perf-time)
        collect (if (and score-begin score-end)
                    (interpolate (lookup-expression old score-begin)
                                perf-time
                                (lookup-expression old score-end)
                                (lookup-expression new score-begin)
                                (lookup-expression new score-end))
                    perf-time))))

```

```

;*****
; mix in to estimate expression in case of absence, by linear inter- or extrapolation
;*****
(defclass estimate-mixin () ())

(defmethod fetch-expression :around ((object musical-object) (expression estimate-mixin))
  (or (get-expression object expression)
      (estimate-expression object expression)))

(defmethod fetch-expression ((object null) (expression expression)) nil)

(defmethod fetch-expression ((object musical-object) (expression expression))
  (get-expression object expression))

(defmethod get-next-expression :around ((object musical-object)
                                         (expression estimate-mixin))
  (cond ((call-next-method))
        ((right object)
         (estimate-expression (right object) expression))
        (t
         (estimate-next-expression object expression))))

(defmethod fetch-onset :around ((object musical-object) (expression estimate-mixin))
  (fetch-expression object (find-expression 'estimate-onset-timing)))

(defmethod estimate-expression ((object musical-object) (expression expression))
  (estimate-context (context-with-expression object expression #'left)
                   object
                   (context-with-expression object expression #'right)
                   expression
                   t))

(defmethod estimate-next-expression ((object musical-object) (expression expression))
  (let* ((left (context-with-expression object expression #'left))
         (left (and left
                     (left left)
                     (context-with-expression (left left) expression #'left))))
    (when (and left left)
      (interpolate (score-onset left)
                   (score-offset object)
                   (score-onset left)
                   (get-expression left expression)
                   (get-expression left expression))))))

(defmethod estimate-context (left object right (expression expression) first-try)
  (cond ((and left right)
        (interpolate (score-onset left)
                     (score-onset object)
                     (score-onset right)
                     (get-expression left expression)
                     (get-expression right expression)))
        ((and left (left left) first-try)
         (estimate-context (context-with-expression (left left) expression #'left)
                          object
                          left
                          expression nil))
        ((and right (right right) first-try)
         (estimate-context right
                          object
                          (context-with-expression (right right) expression #'right)
                          expression nil))
        (t nil)))

(defmethod context-with-expression ((object musical-object)
                                     (expression expression) direction)
  (cond ((get-expression object expression)
        object)
        ((funcall direction object)
         (context-with-expression (funcall direction object) expression direction))
        (t nil)))

```

```

;*****
; keeping articulation invariant: mixin for expressive timing expression
;*****

(defclass keep-articulation-mixin () ())
(defclass keep-overlap-articulation-mixin (keep-articulation-mixin) ())
(defclass keep-duration-articulation-mixin (keep-articulation-mixin) ())
(defclass keep-proportion-articulation-mixin (keep-articulation-mixin) ())

(defmethod articulation ((expression keep-overlap-articulation-mixin))
  (find-expression 'basic-overlap-articulation))

(defmethod articulation ((expression keep-duration-articulation-mixin))
  (find-expression 'basic-duration-articulation))

(defmethod articulation ((expression keep-proportion-articulation-mixin))
  (find-expression 'basic-proportion-articulation))

(defmethod set-map :around ((object musical-object)
                             map
                             (expression keep-articulation-mixin)
                             ground)
  (when map
    (let* ((parts (find-parts object ground))
           (articulation-collections
            (loop for part in parts
                  collect (get-notes-expression part (articulation expression)))))
      (call-next-method)
      (loop for part in parts
            for collection in articulation-collections
            do (set-notes-expression part (articulation expression) collection)))
    object))

;*****
; resource for expression instances

(defvar *expression-instances*)
(setf *expression-instances* nil)
(defvar *use-expression-resource*)
(setf *use-expression-resource* t)

(defun find-expression (class)
  (or (and *use-expression-resource*
          (cdr (assoc class *expression-instances*)))
      (make-expression-instance class)))

(defun make-expression-instance (class)
  (let ((instance (make-instance class)))
    (when *use-expression-resource*
      (push (cons class instance) *expression-instances*)))
  instance)

;*****
; averaging expression
;*****

(defclass averaging-expression-mixin () ())

;*****
; get averaging expression

(defmethod get-expression ((object multilateral) (expression averaging-expression-mixin))
  (loop for component in (components object)
        when (get-expression component expression)
        sum it into total
        finally (return (/ total (length (components object)))))

(defmethod get-expression ((object collateral) (expression averaging-expression-mixin))
  (get-expression (main object) expression))

```

```

;*****
; set averaging expression

(defmethod set-expression ((object multilateral)
                           (expression averaging-expression-mixin)
                           (section multilateral-section))
  (loop for component in (components object)
        for new-expression in (expressions section)
        do (propagate-shift component
                             (save-- new-expression
                                       (fetch-expression component expression))
                             expression)))

(defmethod set-expression ((object collateral)
                           (expression averaging-expression-mixin)
                           (section collateral-section))
  (propagate-shift (ornament object)
                   (save-- (ornament-expression section)
                           (fetch-expression (ornament object) expression))
                   expression))

;*****
; scale averaging expression

(defmethod scale-expression ((section multilateral-section)
                             (expression averaging-expression-mixin)
                             factor)
  (let* ((mean-expression (mean (expressions section)))
         (expression-deviations (mapcar #'(lambda (expression)
                                             (- expression mean-expression))
                                         (expressions section)))
         (new-expressions
          (mapcar #'(lambda (expression-deviation)
                      (+ mean-expression
                        (scale-expression-lin expression-deviation factor)))
                  expression-deviations)))
    (make-new-section section new-expressions)))

(defmethod scale-expression ((section collateral-section)
                             (expression averaging-expression-mixin)
                             factor)
  (let* ((expression-deviation (- (ornament-expression section)
                                   (main-expression section)))
         (new-ornament-expression
          (+ (main-expression section)
             (scale-expression-lin expression-deviation factor))))
    (make-new-section section
                      (list new-ornament-expression
                            (main-expression section)))))

;*****
; stretch averaging expression

(defmethod stretch-expression ((section S-section)
                               (old S-map)
                               (new S-map)
                               (expression averaging-expression-mixin))
  (make-new-section
   section
   (loop for expression in (expressions section)
         for score-time in (score-times section)
         as old-expression = (lookup-expression old score-time)
         as new-expression = (lookup-expression new score-time)
         as stretched-expression = (if (and old-expression new-expression expression)
                                       (+ expression (- new-expression old-expression))
                                       expression)
         collect stretched-expression)))

```

```

;*****
; ARTICULATION
;*****

(defclass offset-timing      (expressive-timing) ())
(defclass articulation      (offset-timing averaging-expression-mixin) ())
(defclass basic-overlap-articulation (articulation) ())
(defclass basic-duration-articulation (articulation) ())
(defclass basic-proportion-articulation (articulation) ())

;*****

(defmethod get-expression ((object NOTE) (expression offset-timing))
  (perf-offset object))

(defmethod fetch-onset ((object musical-object) (expression articulation))
  (get-expression object (find-expression 'onset-timing)))

;*****
; get articulation

(defmethod get-expression ((object NOTE) (expression basic-overlap-articulation))
  (when (right object)
    (save-- (perf-offset object)
            (fetch-onset (right object) expression))))

(defmethod get-expression ((object NOTE) (expression basic-duration-articulation))
  (- (perf-offset object)
     (fetch-onset object expression)))

(defmethod get-expression ((object NOTE) (expression basic-proportion-articulation))
  (when (and (fetch-onset object expression)
              (right object)
              (fetch-onset (right object) expression))
    (/ (- (perf-offset object)
          (fetch-onset object expression))
       (- (fetch-onset (right object) expression)
          (fetch-onset object expression)))))

;*****
; set articulation

(defmethod set-expression ((object NOTE) (expression basic-overlap-articulation) value)
  (when (and (right object) (fetch-onset (right object) expression))
    (setf (perf-offset object)
          (max (fetch-onset object expression)
               (+ (fetch-onset (right object) expression)
                  value)))))

(defmethod set-expression ((object NOTE) (expression basic-duration-articulation) value)
  (setf (perf-offset object)
        (+ (fetch-onset object expression)
           (max 0 value))))

(defmethod set-expression ((object NOTE)
                           (expression basic-proportion-articulation) value)
  (when (and (right object) (perf-onset (right object)))
    (setf (perf-offset object)
          (+ (fetch-onset object expression)
             (* (- (fetch-onset (right object) expression)
                  (fetch-onset object expression))
                (max 0 value))))))

```



```

;*****
; empty expression (to recover only score times)
;*****

(defclass empty-expression (expression) ())

(defmethod get-expression ((object musical-object) (expression empty-expression)) nil)

;*****
; mixing instantiable classes of expression
;*****

(defmacro class-mixer (&rest class-cocktail-pairs)
  (list* 'progl t
    (loop for tuples on class-cocktail-pairs by #'cdddr
      as name = (first tuples)
      as doc = (second tuples)
      as cocktail = (third tuples)
      collect `(defclass ,name ,cocktail ()
        (:documentation ,doc)))))

```

```

(class-mixer
  tempo " "
  (basic-tempo)

  asynchrony " "
  (basic-asynchrony)

  estimate-tempo " "
  (basic-tempo estimate-mixin)

  estimate-asynchrony " "
  (basic-asynchrony estimate-mixin)

  keep-overlap-articulation-tempo " "
  (basic-tempo keep-overlap-articulation-mixin)

  keep-duration-articulation-tempo " "
  (basic-tempo keep-duration-articulation-mixin)

  keep-proportion-articulation-tempo " "
  (basic-tempo keep-proportion-articulation-mixin)

  keep-overlap-articulation-estimate-tempo " "
  (basic-tempo keep-overlap-articulation-mixin estimate-mixin)

  keep-duration-articulation-estimate-tempo " "
  (basic-tempo keep-duration-articulation-mixin estimate-mixin)

  keep-proportion-articulation-estimate-tempo " "
  (basic-tempo keep-proportion-articulation-mixin estimate-mixin)

  keep-overlap-articulation-asynchrony " "
  (basic-asynchrony keep-overlap-articulation-mixin)

  keep-duration-articulation-asynchrony " "
  (basic-asynchrony keep-duration-articulation-mixin)

  keep-proportion-articulation-asynchrony " "
  (basic-asynchrony keep-proportion-articulation-mixin)

  keep-overlap-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-overlap-articulation-mixin estimate-mixin)

  keep-duration-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-duration-articulation-mixin estimate-mixin)

  keep-proportion-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-proportion-articulation-mixin estimate-mixin)

  overlap-articulation " "
  (basic-overlap-articulation)

  duration-articulation " "
  (basic-duration-articulation)

  proportion-articulation " "
  (basic-proportion-articulation)

  estimate-overlap-articulation " "
  (basic-overlap-articulation estimate-mixin)

  estimate-duration-articulation " "
  (basic-duration-articulation estimate-mixin)

  estimate-proportion-articulation " "
  (basic-proportion-articulation estimate-mixin))

```

```

;*****
;*****
; EXTRACTING AND IMPOSING EXPRESSION MAPS OF MUSICAL OBJECTS USING EXPRESSION
;*****
;*****
; extracting a expression map

(defmethod get-map ((object musical-object) expression ground)
  (make-map (loop for part in (find-parts object ground)
                  collect (get-section part expression))))

(defmethod get-section ((object musical-object) expression)
  (make-section (object-to-section object)
                (snoc (mapcar #'score-onset (components object))
                      (score-offset object))
                (snoc (mapcar #'(lambda (component)
                                   (fetch-expression component expression))
                          (components object))
                      (get-next-expression object expression))))

;*****
; impose a expression map

(defmethod set-map ((object musical-object) map expression ground)
  (loop for part in (find-parts object ground)
        for section in (sections map)
        do (set-expression part expression section))
  object)

```

```

;*****
;*****
; OPERATIONS ON EXPRESSION MAPS
;*****
;*****
; scale expression map

(defmethod scale-map ((map map) expression factor)
  (with-filtered-null-expression #'(lambda (filtered-map)
    (scale-filtered-map filtered-map expression factor))
    map))

(defmethod scale-filtered-map ((map map) expression factor)
  (map-map #'(lambda (section)
    (scale-expression section
      expression
      (get-parameter factor (score-onset section))))
    map))

;*****
; interpolate S-expression maps

(defmethod interpolate-maps ((map1 S-map) (map2 S-map) factor)
  (map-map #'(lambda (section) (interpolate-section section
    (filter-null-expression map2)
    factor))
    map1))

(defmethod interpolate-section ((section S-section) (map S-map) factor)
  (make-new-section
    section
    (loop for score-time in (score-times section)
      for expression in (expressions section)
      collect (in-between expression
        (lookup-expression map score-time)
        (get-parameter factor score-time)))))

(defmethod monotonise-map ((map S-map))
  (map-map #'monotonise-section map))

(defmethod monotonise-section ((section S-section))
  (make-new-section
    section
    (loop for expression in (expressions section)
      when expression
      maximize expression into state
      and collect state
      else collect nil)))

;*****
; get S-expression maps at sync points

(defmethod get-sync-map ((map1 S-map) (map2 S-map))
  (map-map #'(lambda (section) (get-sync-section section map2)) map1))

(defmethod get-sync-section ((section S-section) (map S-map))
  (make-new-section-from-pairs section
    (loop for score-time in (all-score-times section)
      for expression in (all-expressions section)
      as new-expression = (and expression
        (lookup-defined-expression map score-time))
      when new-expression collect (list score-time expression))))

```

```

;*****
; stretch expression map

(defmethod stretch-map ((map successive-map)
                        (old successive-map)
                        (new successive-map)
                        expression)
  (let ((filtered-map (filter-null-expression map))
        (filtered-old (filter-null-expression old))
        (filtered-new (filter-null-expression new))
        (removed (filter-null-expression-out map)))
    (unfilter-null-expression
     (map-map
      #'(lambda (section)
          (stretch-expression section filtered-old filtered-new expression))
      filtered-map)
     removed)))

;*****
;*****
; TIME-CHANGING PARAMETERS
;*****
;*****

(defun get-parameter (factor score-time)
  (if (numberp factor)
      factor
      (funcall factor score-time)))

(defun make-ramp (x1 x2 y1 y2) ; as s-section ??
  #'(lambda (x) (interpolate x1 x x2 y1 y2)))

```

```

;*****
;*****
; TRANSFORMATIONS ON MUSICAL OBJECTS
;*****
;*****
; transfer expression transformation

(defmethod transfer ((object musical-object) expression foreground background)
  (let* ((foreground-map (get-map object expression foreground))
         (background-map (get-map object (find-expression 'empty-expression) background))
         (new-background-map (interpolate-maps background-map foreground-map 1)))
    (set-map object new-background-map expression background))
  object)

;*****
; scale expression transformation

(defmethod scale ((object musical-object) expression foreground background factor)
  (let* ((old-foreground-map (get-map object expression foreground))
         (new-foreground-map (when old-foreground-map
                                (scale-map old-foreground-map expression factor)))
         (old-background-map (when background
                                (get-map object expression background)))
         (new-background-map (when old-background-map
                                (stretch-map old-background-map
                                              old-foreground-map
                                              new-foreground-map
                                              expression))))
    (when new-foreground-map
      (set-map object new-foreground-map expression foreground))
    (when new-background-map
      (set-map object new-background-map expression background)))
  object)

;*****
; scale intervoice expression transformation

(defmethod scale-intervoice ((object musical-object) expression
                             voice1 voice2 factor ref)
  (let* ((map1 (get-map object expression voice1))
         (map2 (get-map object expression voice2)))
    (when (and map1 map2)
      (let* ((original-sync-map1 (get-sync-map map1 map2))
             (original-sync-map2 (get-sync-map map2 map1))
             (new-sync-map1 (monotonise-map (interpolate-maps
                                              original-sync-map1
                                              original-sync-map2
                                              (* ref (- 1 factor))))))
        (new-sync-map2 (monotonise-map (interpolate-maps
                                          original-sync-map2
                                          original-sync-map1
                                          (* (- 1 ref) (- 1 factor))))))
        (new-map1 (stretch-map
                    map1 original-sync-map1 new-sync-map1 expression))
        (new-map2 (stretch-map
                    map2 original-sync-map2 new-sync-map2 expression)))
      (set-map object new-map1 expression voice1)
      (set-map object new-map2 expression voice2)))
  object))

```

```

;*****
;*****
; LISP UTILITIES
;*****
;*****

(defun last-element (list)
  (first (last list)))

(defun snoc (list item)
  (append list (list item)))

(defun mean (numbers)
  (/ (apply #'+ numbers) (length numbers)))

(defun save-min (&rest list)
  (let ((new-list (remove nil list)))
    (and new-list (apply #'min new-list))))

(defun save-max (&rest list)
  (let ((new-list (remove nil list)))
    (and new-list (apply #'max new-list))))

(defun save-- (&rest list)
  (and (notany #'null list)
    (apply #'- list)))

(defun save+ (&rest list)
  (apply #'+ (remove nil list)))

(defun enforce-limits (minimum x maximum)
  (max minimum (min x maximum)))

(defun integrate (list start)
  (if (null list)
    (list start)
    (cons start
      (integrate (rest list) (+ (first list) start))))))

(defun normalise (list dur)
  (let ((factor (/ dur (apply #'+ list))))
    (mapcar #'(lambda (item) (* factor item)) list)))

(defun interpolate (x1 x x2 y1 y2)
  (cond ((eql y1 y2) y1)
        ((eql x1 x2) nil)
        ((null x) nil)
        ((and x1 (= x x1)) y1)
        ((and x2 (= x x2)) y2)
        ((and x1 x2)
         (in-between y1 y2 (/ (- x x1) (- x2 x1))))
        (t nil)))

(defun in-between (y1 y2 a)
  (cond ((= a 0) y1)
        ((= a 1) y2)
        ((and y1 y2)
         (+ y1 (* a (- y2 y1))))
        (t nil)))

```

```

;*****
;*****
; EXAMPLES
;*****
;*****
#|

(defun metre-example ()
  (S 'bars
    (P 'bar
      (S 'melody
        (PAUSE :name 'pause :score-dur 1/4)
        (NOTE :name 64 :score-dur 1/8
          :perf-onset .30 :perf-offset 0.5 :dynamic .7))
      (S 'accompaniment
        (PAUSE :name 'pause :score-dur 3/8)))
    (P 'bar
      (S 'melody
        (APPOG 'appoggiatura
          (NOTE :name 64 :score-dur 1/8
            :perf-onset .550 :perf-offset .680 :dynamic .75)
          (NOTE :name 55 :score-dur 1/4
            :perf-onset .675 :perf-offset 1.133 :dynamic .7))
        (NOTE :name 55 :score-dur 1/8
          :perf-onset 1.125 :perf-offset 1.475 :dynamic .7))
      (S 'accompaniment
        (NOTE :name 38 :score-dur 1/8
          :perf-onset .725 :perf-offset .90 :dynamic .6)
        (NOTE :name 43 :score-dur 1/8
          :perf-onset .95 :perf-offset 1.2 :dynamic .6)
        (NOTE :name 47 :score-dur 1/8
          :perf-onset 1.150 :perf-offset 1.475 :dynamic .7)))
    (P 'bar
      (S 'melody
        (ACCIA 'acciaccatura
          (NOTE :name 59 :score-dur 1/16
            :perf-onset 1.600 :perf-offset 1.7 :dynamic .65)
          (NOTE :name 57 :score-dur 1/8
            :perf-onset 1.625 :perf-offset 1.880 :dynamic .7))
        (NOTE :name 55 :score-dur 1/8
          :perf-onset 1.880 :perf-offset 2.256 :dynamic .6)
        (NOTE :name 57 :score-dur 1/8
          :perf-onset 2.256 :perf-offset 2.647 :dynamic .65))
      (S 'accompaniment
        (P 'chord
          (NOTE :name 38 :score-dur 3/8
            :perf-onset 1.725 :perf-offset 2.500 :dynamic .7)
          (NOTE :name 42 :score-dur 3/8
            :perf-onset 1.775 :perf-offset 2.500 :dynamic .65)
          (NOTE :name 48 :score-dur 3/8
            :perf-onset 1.800 :perf-offset 2.500 :dynamic .7))))
    (P 'bar
      (S 'melody
        (NOTE :name 55 :score-dur 3/8
          :perf-onset 2.425 :perf-offset 4 :dynamic .7))
      (S 'accompaniment
        (P 'chord
          (NOTE :name 43 :score-dur 3/8
            :perf-onset 2.500 :perf-offset 4 :dynamic .6)
          (NOTE :name 47 :score-dur 3/8
            :perf-onset 2.550 :perf-offset 4 :dynamic .7)
          (NOTE :name 50 :score-dur 3/8
            :perf-onset 2.580 :perf-offset 4.5 :dynamic .65))))))

```



```

(defun background-example ()
  (P 'fragment
    (S 'melody
      (PAUSE :name 'pause :score-dur 1/4)
      (NOTE :name 64 :score-dur 1/8
        :perf-onset 0.3 :perf-offset 0.5 :dynamic .7)
      (APPOG 'appoggiatura
        (NOTE :name 64 :score-dur 1/8
          :perf-onset .550 :perf-offset .680 :dynamic .75)
        (NOTE :name 55 :score-dur 1/4
          :perf-onset .675 :perf-offset 1.133 :dynamic .7))
      (NOTE :name 55 :score-dur 1/8
        :perf-onset 1.125 :perf-offset 1.475 :dynamic .7)
      (ACCIA 'acciaccatura
        (NOTE :name 59 :score-dur 1/16
          :perf-onset 1.600 :perf-offset 1.700 :dynamic .65)
        (NOTE :name 57 :score-dur 1/8
          :perf-onset 1.625 :perf-offset 1.880 :dynamic .7))
      (NOTE :name 55 :score-dur 1/8
        :perf-onset 1.880 :perf-offset 2.256 :dynamic .6)
      (NOTE :name 57 :score-dur 1/8
        :perf-onset 2.256 :perf-offset 2.647 :dynamic .65)
      (NOTE :name 55 :score-dur 3/8
        :perf-onset 2.425 :perf-offset 4 :dynamic .7))
    (S 'accompaniment
      (PAUSE :name 'pause :score-dur 3/8)
      (NOTE :name 38 :score-dur 1/8
        :perf-onset .725 :perf-offset .90 :dynamic .6)
      (NOTE :name 43 :score-dur 1/8
        :perf-onset .950 :perf-offset 1.2 :dynamic .6)
      (NOTE :name 47 :score-dur 1/8
        :perf-onset 1.150 :perf-offset 1.475 :dynamic .7)
      (P 'chord
        (NOTE :name 38 :score-dur 3/8
          :perf-onset 1.725 :perf-offset 2.500 :dynamic .7)
        (NOTE :name 42 :score-dur 3/8
          :perf-onset 1.775 :perf-offset 2.500 :dynamic .65)
        (NOTE :name 48 :score-dur 3/8
          :perf-onset 1.800 :perf-offset 2.500 :dynamic .7))
      (P 'chord
        (NOTE :name 43 :score-dur 3/8
          :perf-onset 2.500 :perf-offset 4 :dynamic .6)
        (NOTE :name 47 :score-dur 3/8
          :perf-onset 2.550 :perf-offset 4 :dynamic .7)
        (NOTE :name 50 :score-dur 3/8
          :perf-onset 2.580 :perf-offset 4.5 :dynamic .65))))))

;data at factor 2 in figure 13
(scale (metre-example)
  (find-expression 'tempo) (has-name? 'bars) nil
  2)

;data at factor 2 in figure 14
(scale (metre-example)
  (find-expression 'asynchrony) (has-name? 'bar) nil
  2)

;data at factor 2 in figure 16a
(scale (background-example)
  (find-expression 'tempo) (has-name? 'melody) nil
  2)

;data at factor 2 in figure 16b
(scale (background-example)
  (find-expression 'tempo) (has-name? 'melody) (has-name? 'accompaniment)
  2)

|#

```

IV

TWO EXAMPLES OF LARGER SYSTEMS

IV

TWO EXAMPLES OF LARGER SYSTEMS

**POCO:
AN ENVIRONMENT FOR
ANALYSING, MODIFYING, AND GENERATING
EXPRESSION IN MUSIC**

Henkjan Honing

© copyright 1990, Henkjan Honing

Published as: Honing, H. (1990). POCO: An Environment for Analysing, Modifying, and Generating Expression in Music. In *Proceedings of the 1990 International Computer Music Conference*. San Francisco: Computer Music Association.

CONTENTS

Introduction.....3

Dreams and visions.....3

Design.....5

I/O.....5

Data representation.....8

 Musical objects8

 Structure.....8

User interface.....8

Transformations.....9

 A typical path.....9

Analyses10

Conclusion10

Acknowledgments.....11

References11

POCO: AN ENVIRONMENT FOR ANALYSING, MODIFYING, AND GENERATING EXPRESSION IN MUSIC

Henkjan Honing

POCO is a workbench for analysing, modifying and generating expression in music. It is aimed to use in a research context. A consistent and flexible representation of musical objects and structure was designed. The integration of existing models of expression made it possible to compare and combine these models using the same performance and score data. New tools were developed for specific "micro surgery" on expression. A lot of attention was given to the openness, integration, and extendibility of the system.

INTRODUCTION

As part of our research on the modelling of expression in musical performance at City University, London, we developed a workbench named POCO. It consists of a collection of tools that can be used for the analysis, modification, and generation of expression in a research context. The research project combines three perspectives: musicological aspects (what are the rules of expression used in different styles of music), cognitive aspects (how does a good performance or interpretation facilitate the understanding of the music by the listener), and computational aspects (the design of appropriate data structures and development of programs dealing with expression). The latter will be described in this paper.

Before describing the system, the next section will give the reader a flavour of some of the problems and ideas related to this research.

DREAMS AND VISIONS

When starting a project like this there are various directions one might take. This is the moment to fantasize about the ideal system, as later on one's thoughts will probably tend to be directed by their feasibility. We will sum up a collection of dreams that we would like to see realised.

First of all, the system should incorporate existing computational models related to expression in music. These should share the same data structures so that they can be evaluated and compared. Combining these models should also be possible.

We are interested in studying a number of issues in expressive performances. For instance, how do the magnitudes, that are used in the different expressive parameters, behave in time and at different tempi, and how do they relate to the musical structure (Clarke 1988). Because a listener cannot detect all the subtle expressive details of a performance we need some help. We envisage the possibility of "zooming-in" at the different structural levels of a musical performance (e.g. examining the expressive timing only at bar level or only at phrase level), as well as looking at its various structural units (e.g. chords or grace notes) or inter-structural relations (e.g. voice leading).

Besides analysis, we would like to perform "micro surgery" on performances: change expressive detail and shape of structural units, or, in other words, generate modified performances that have been transformed depending on particular structural units. To give some examples: we would like to exaggerate the timing of chords without changing their spread (or the reverse), change the tempo of a piece without altering the timing of grace notes and trills, modify the timing of the melody without changing the timing of the accompaniment, remove all expressive timing except on beat level, scale specific structural elements of a performance using different magnitudes, or make a solo voice lead with respect to the rest of the music. The results of these adjustments (i.e. modified performances) could then be used in experiments where listeners have to judge the modified performance on the basis of their perceptual effectiveness.

In order to study expression in a performance a score is essential. When scores are not available (in the case of e.g. improvisations) we are helped by an automated score generator.

When scores are available, computer assistance is indispensable in mutually adjusting the performance and the score (e.g. taking care of performance errors, order of notes within chords, ornaments in the performance etc.), since we have to compare them on a note-to-note basis. It should also assist in transferring structural information from the score to the performance (e.g. left and right hand parts), instead of having to annotate each new performance.

Of course the possibility of recording and playing back performances of different types of instruments is an important requirement, next to having access to libraries of (expert) performances and scores, and employing graphical and textual editors in editing the musical and structural information.

All the means described should be embedded in a programming environment in order to gain maximum flexibility and extendibility. The environment should support version management (keep track of different versions of data and how it was created), assistance in repetitive work (e.g. when doing the same analysis over all the data of an experiment),

and the automatic generation of documentation about the system. These are just a few demands on system support.

Finally, both first-time and advanced users should feel comfortable working with the system. First time users should be able to make use of menu's and dialogs, and have explanatory information on the actions that are performed. More advanced users probably want to bypass the menu's and dialogs using a programmed way of manipulation. The user interface should be multi-modal, both simple and flexible, and it should be easy for advanced users to extend the environment and have their programs well integrated.

We will try to give shape to this hotchpotch of dreams and visions in the following paragraphs. The described "ideal" system is simplified into a conceptual description in Figure 1.

DESIGN

Earlier work on composition systems (Desain & Honing 1988) gave us enough confidence in the importance of building POCO by using a workbench approach: a collection of tools that can be combined in a flexible way. This resulted in an architecture that embodies a relatively empty shell consisting of a closed data representation at one end and the user interface at the other. In between there is a layer of commands (or transformations) that is extendible. Communication with the outside world (e.g. sequencers and statistical packages) is supported by an i/o layer and is extendible as well (e.g. when a new medium is added or a new format is needed). This architecture is shown in Figure 2.

In the remaining half of the article we will describe this architecture layer by layer.

I/O

Communication with the outside world is implemented as transparent as possible and is modelled as streams, a combination of a medium (e.g. a file, a window, a Midi-port) and its associated i/o-type(s) (e.g. formats, protocols). The system provides different i/o-types (e.g. music-text-files, standard Midi files). All information generated by the system is encoded in the specific format or protocol used, so there is always completeness of information. A new medium and its i/o-type(s) can easily be added by providing a set of read and write functions.

We support the Midi standard to be able to use commercial software for capturing and play-back, facilitating the exchange of performances and scores between systems, and making use of the growing range of Midi based instruments and interfaces. The format was extended to sustain completeness of information. Within the system the musical information is encoded into a more general data representation.

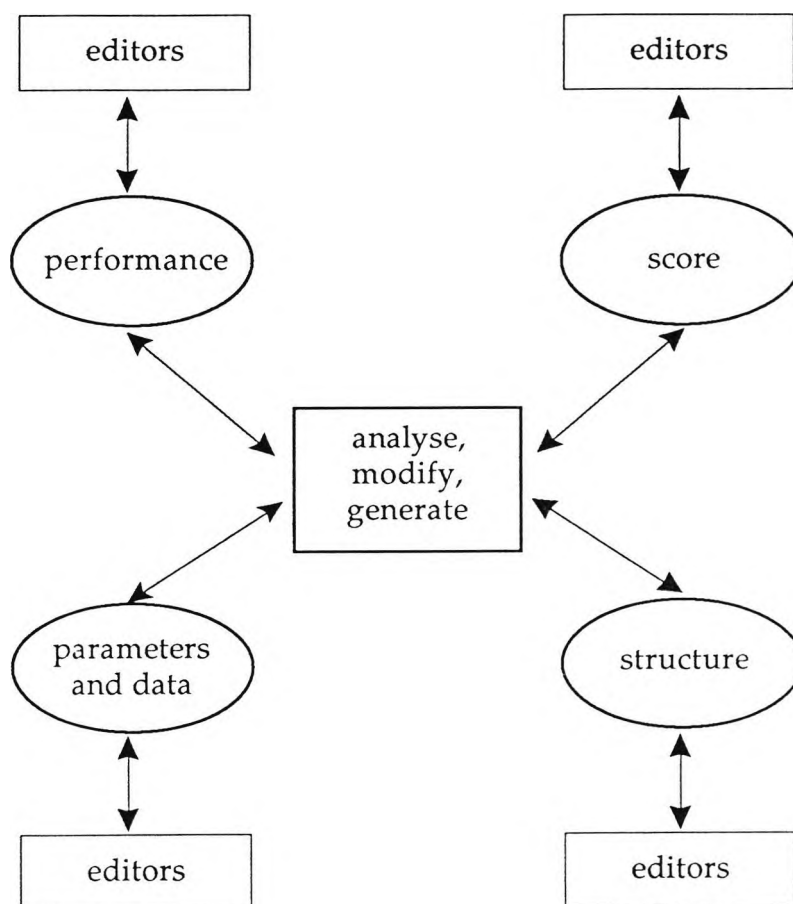


Figure 1. Conceptual design.

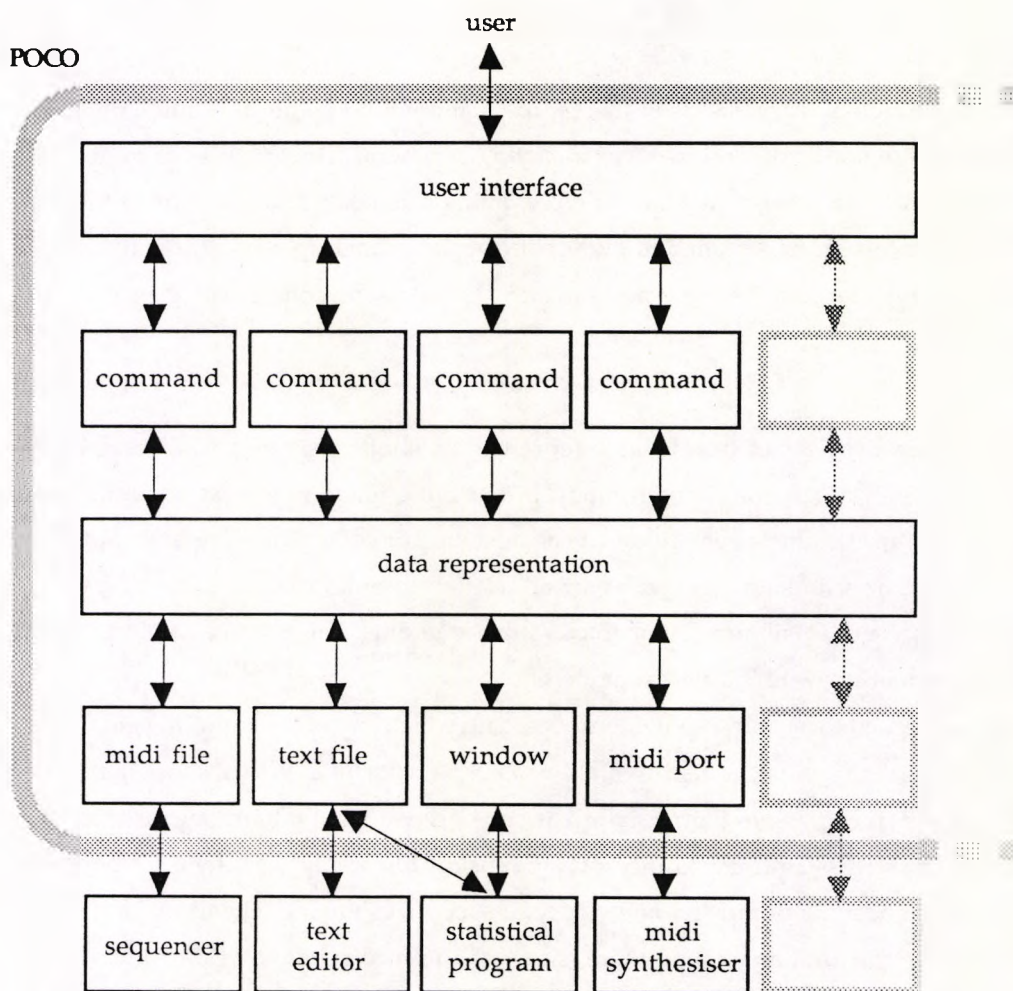


Figure 2. Functional design.

DATA REPRESENTATION

A consistent and flexible representation of musical objects within the environment is essential because all operations take place on this representation.

Musical objects

There are two kinds of basic objects in this representation: time-points and time-intervals. Time-intervals are note, rest, and segment (denoting structure). Time-points are midi (e.g. Midi controller information), comment (for representing comments and other timed textual information), and begin-of-stream and end-of-stream (to model upbeats, to calculate the length of a piece, to cut sections out of performances, to merge and concatenate them etc.).

Structure

One of the main deficiencies of low level representations of music (e.g. Midi files, note lists) is the absence of structural descriptions. In our representation we use a simple and flexible way of representing structure called segmentation or collections. The basic musical objects can be grouped using a general 'part-of' relation to build hierarchical, horizontal, vertical, associative or even mutual ambiguous structural units. This representation proved to suffice in rebuilding wildly different models.

Each unit is named to be able to provide a hook onto which any other knowledge (outside the definition of the musical representation) can be attached. When constructing a complete model of expressive timing, information is needed from a harmonic or metrical nature. Although it is tempting to incorporate musical knowledge, as done in most AI approaches to modelling of musical knowledge, it specializes the model and makes it less modular. With structural annotation there is no need to incorporate all this domain (i.e. style) specific information in the system, because it can be communicated through a layer of structural information (see also Honing 1990).

USER INTERFACE

POCO is implemented in Allegro Common Lisp making use of program generators. They facilitate the easy integration of user code. When a new command is added to the system, it automatically propagates information to the right menu's and dialogs and provides information for the automatic documentation generator (a facility that is almost indispensable in a larger system).

The user interface supports multiple modes of communication, consisting of menu's and dialogs, Lisp program equivalents, and natural language descriptions (see figure 3). The

system keeps a history of all actions that took place. They are available as normal Lisp expressions that can be re-evaluated and edited. Data files generated by the system contain information describing what transformations were used and their parameters (i.e. the Lisp expression that generated it).

TRANSFORMATIONS

Transformations are a (still growing) collection of tools that generate new or modified musical information. There is a matcher for comparing, cleaning-up, and mutually adjusting scores and performances, a filter system (using a general pattern language) to retrieve special information (e.g. all notes that are part of a chord, the whole piece except the ornaments, all notes in the left hand of a piano performance in the second phrase etc.), tools that allow scaling of timing, articulation and dynamics of musical objects (e.g. amplifying, translating or inverting the expressive timing profiles), and transformations to merge or concatenate performances or scores.

Another set of tools embodies some well-known models of expression. Longuet-Higgins' metrical parser (1987), Todds model of rubato (1989), the Sundberg (1989) expression generating rule system (Van Oosten 1990), and the Desain and Honing connectionist quantizer (1989) are examples of transformations that are available.

A typical path

To give an idea of both the complexity of an expressive transformation, which might seem simple at first sight, and the support given by the system in the realisation of such transformation, we will describe a typical path from an original piano performance to a new version with a modified expressive timing profile depending on the musical structure.

To be able to look at the expressive timing of the performance we need a score. Either we use a score available in one or the other standard formats or we can make a new one from a recorded performance using one of the quantizers (Desain & Honing 1989; Longuet-Higgins 1987) resulting in a first version of the score. Then we probably need to do some editing of the score, for instance, add (more) structural information, correct errors etc. This can be done by using the editors outside the system, after converting the score to a convenient format. But before we can do any transformation the score should be matched to the performance under examination (removing errors in the performance, altering order of notes within chords etc.). All non-note (e.g. rests, comments) and structural information annotated in the score is merged into the performance and vice versa. The result is a matched performance and score, both with all the available structural information. These form the basic input to our transformations. We can now, for instance, exaggerate the timing of the bars,

without changing the timing of the other structural units (e.g. chords and phrases). The modified performance is written to an external file that can be played by a sequencer.

The environment offers different kinds of help that makes traveling along this path, with all its intermediate steps, easier and repeatable (see User Interface).

ANALYSES

Analysis is a category of transformations that generates statistical data (instead of musical information). The analyses comprises special analytical methods that provide the user with textual or numerical information. It will be written to a selected medium (e.g. file, window) and can be used by programs outside the system (e.g. statistical packages), were additional analysis can be done, graphs can be plotted etc.

Examples of analyses provided are the use of autocorrelation in analysing expressive timing (Desain & De Vos 1990), analysis that produce tables of timing data related to structure that facilitate the study of e.g. voice-leading and chord timing. These are among other more straightforward analyses.

CONCLUSION

Although we, of course, didn't succeed in realising all the dreams of a ideal system, we provided a sound basis for further development. We do think to have made the right decisions in what should be inside and what outside the system. The possibility to use structure in examining and manipulating expression proved to be very powerful. The facilities described in User Interface turned out not to be just luxurious, but improved the usability and maintainability of the system.

POCO is currently used by the institutes involved in the project, but is still in development. We now work on stabilising the system. A version for distribution is not yet available.

Our own use of the system is directed towards understanding the relation between expression and structure, hopefully resulting in more insights of how to model expression in music. This would enable us to design editors that can manipulate musical information in a more psychologically and perceptually relevant way. In the end we hope to contribute to the design of composition and interactive computer systems in need of models for the production and perception of musical performance.

ACKNOWLEDGMENTS

Both design and realisation of POCO was done in collaboration with Peter Desain. The research was done together with Eric Clarke at City University, London and was made possible by an ESRC grant under number A413254004.

Thanks to Steve McAdams for fruitful discussions during the development of the system. Also thanks to Peter van Oosten, Klaus de Rijk, Jeroen Schuit, Siebe de Vos, and our other colleagues at the Centre for Knowledge Technology for their help and advice. And especially Johan den Biggelaar, Ton Hokken and Thera Jonker for their special support.

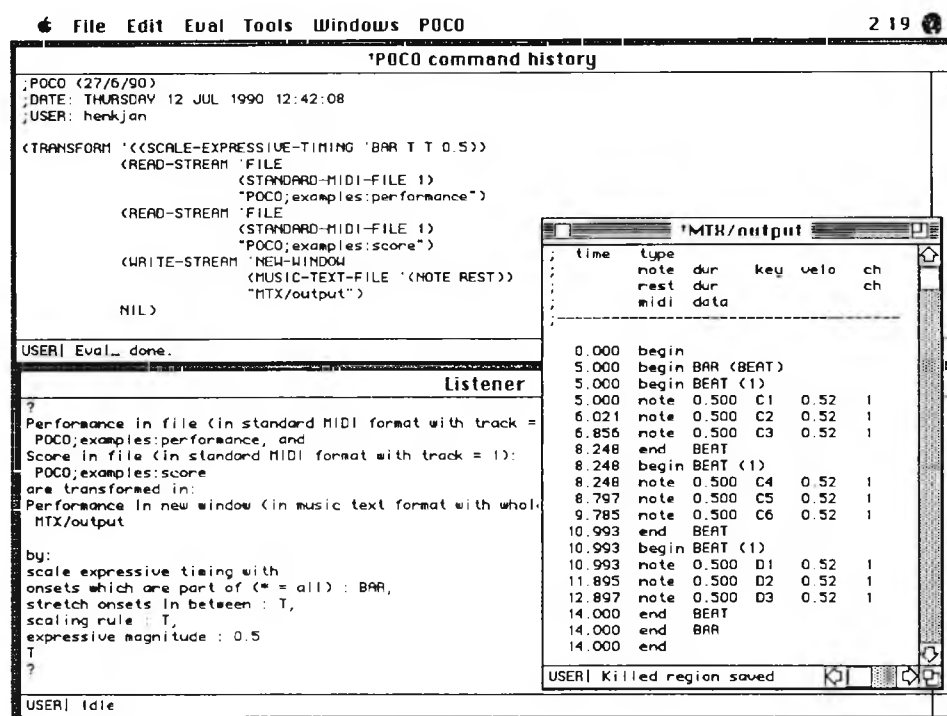


Figure 3. A snapshot of the system.

REFERENCES

- Clarke, E.F. 1988. "Generative principles in music performance." In J. Sloboda (Ed.) *Generative Processes in Music*. Oxford: The Clarendon Press.
- Desain, P. and H. Honing. 1988. "LOCO: A Composition Microworld in Logo" *Computer Music Journal* 12(3), Cambridge, Mass.: MIT Press.
- Desain, P. and H. Honing. 1989. "Quantization of Musical Time: A Connectionist Approach." *Computer Music Journal* 13(3). Cambridge, Mass.: MIT Press.

- Desain, P. and S. de Vos. 1990. "Autocorrelation and the Study of Musical Expression." In: Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association.
- Honing, H. 1990. "Issues in the Representation of Time and Structure in Music". To be presented at the Music and the Cognitive Sciences Conference, 16-21 September 1990, Cambridge.
- Longuet-Higgins, H.C. 1987. Mental Processes. Cambridge, Mass.: MIT Press.
- Oosten, P. van. 1990. "A Critical Study of Sundberg's Rules for Expression in the Performance of Melodies". To be presented at the Music and the Cognitive Sciences Conference, 16-21 September 1990, Cambridge.
- Thompson, W., J. Sundberg, A. Friberg, and L. Frydén. 1989. "The Use of Expression in the Performance of Melodies" *Psychology of Music and Music Education* 17.
- Todd, N. 1989. "A Computational Model of Rubato" In: "Music, Mind and Structure". E.F. Clarke and S. Emmerson (Eds.) *Contemporary Music Review* 3(1).

Peter Desain and Henkjan Honing

LOCO Foundation
P.O. Box 1037
NL-3500 BA Utrecht, The Netherlands

Center for Art, Media, and Technology
Utrecht Academy of Arts
P.O. Box 1520
NL-3500 BM Utrecht, The Netherlands

LOCO: A Composition Microworld in Logo

Introduction

Ideally, a system for music composition should enable a composer to express ideas in a direct way. Most such systems are limited to a certain style of music, using known aesthetics. These programs are not very useful for creative purposes. It is, of course, impossible to implement all musical criteria and rules one can think of. If the focus is not on a music production system but on a composition system, things become easier. Such a system must be:

User extendible: the user can define new functions that have the same status and possibilities as the already provided, built-in ones.

Modular: functions and objects are isolated and protected from each other and can be studied and used separately.

Orthogonal: functions and objects can be represented in such a way that any future extension will still fit in the system and make use of all the features.

The system should also have these qualities:

Good mechanisms for naming: coding of musical parameters and objects is as close as possible to normal use (so mezzoforte instead of "67," if the user is accustomed to these terms, or -10 db, or "ear-splitting," in their respective cases).

Good mechanisms for abstraction: allowing families of related objects to be defined, differing in one or more parameters.

These design objectives closely resemble the objec-

tives for a general programming language, and indeed a good language would suffice. But to provide only a general programming language, thus asking composers to become programmers, is not reasonable. The present generation of computer music composers acknowledge the cumbersome indirectness of expressing ideas in programs, especially when working, out of habit or need of calculation speed, with low-level languages.

There is another approach. By providing a set of powerful tools in an integrated composition environment, composers are freed from the need to program ad hoc solutions for their ideas. In special cases they, of course, must be able to make use of the power of a programming language, but then the tools are still of great help. In this way the distinction between programming and using already existing programs disappears.

The tools provided take the form of *program generators*, which do much of the work for the composer. To be able to implement program generators (a common practice in artificial intelligence research) we needed a symbolic language like Lisp (Anderson, Corbett, and Reiser 1987). Logo, based on Lisp, would also suffice (Harvey 1985). This language, however, had the enormous advantage of being easy to learn, enabling composers to write directly in Logo. It is also widely available on different microcomputers. This gives its creative possibilities to a much broader user group (Beckwith 1975). A final reason to choose Logo was that it enabled us to use machines at both ends of the processing-power spectrum, from Lisp machines to small personal computers. Development work could be done with all the power of the programming environment of the Lisp machine, while we are still able to port the programs to cheaper computers.

Computer Music Journal, Vol. 12, No. 3, Fall 1988,
© 1988 Massachusetts Institute of Technology.

Music System Architecture

At the end of the 1950s and in the early 1960s the first computer music systems came into existence. One of the first revolutionary aspects of these experiments were the algorithmic composition programs. The computer-generated score had to be played by human musicians. But soon computers became fast enough to generate sound themselves. This was such a fascinating aspect of computer music that it took about a decade before a revival of interest in the compositional processes reappeared (of course there were a few happy exceptions). Profiting from the development of higher, more abstract programming languages, compositional systems were designed (Koenig 1971; Berg 1979; Fry 1984; Jones 1981; Schottsteadt 1983).

The goal of defining any architecture—any system design—should be splitting complex problems into simpler ones. Although often real world systems are only nearly decomposable, a system designer should look carefully where to make an artificial division—or decomposition—of a system.

In general most music can, from a production viewpoint, be looked at from two sides: the view of the composer and the view of the interpreter. Or to put it in another way, the composition and the instrument. The composer makes a piece according to all sorts of rules, conventions, intuitions, and tastes. A composition can result in a score meant as the best, or one of several possible representations of the compositional work. A score could be anything left over from the composition phase for the interpreter or instrument, ranging from codes to control a signal processor to a traditional printed score. This score is decoded by the interpreter with more or less freedom into an audible form. In computer music such a division in three (composition, score, and instrument) turns out to be a convenient one. It is often used implicitly. So one can define three languages. The language of the composition system, the language in which the resulting score is expressed, and the instrument language. These languages can vary between a general programming language and a simple data representation. The information flow can be in both directions, as is the case with interactive composing and music analysis.

Architecture of the Loco System

Our first concern was to design a flexible score language that would always behave as a convenient communication vehicle between any devisable composition and instrument system. Given enough power, the score language can be used for expressing intermediate composition results, and can function as the main data representation in the compositional part of the system. It should indeed be a general representation language for musical objects.

The Score: Representation of Musical Objects

This language is described as a set of primitives (basic building blocks) and ways of structuring small objects into bigger ones. In defining languages one is often tempted to start creating primitives. This is reflected in the huge lists of features often presented in programming language advertisements. But general and powerful ways to build objects out of smaller ones is the central issue. So in defining a score language we chose not to prescribe any primitives, but we do provide the mechanisms for creating them. Anything an instrument system can handle on its own, will, by definition, be a primitive musical object. This ensures that the compositional system is as much as possible independent of the instrument, in the way that good programs are machine independent. Let's first give some examples of interesting instruments and their primitives.

Primitives

If the instrument system is an interface to a mechanical percussion installation, it will know about materials, place, and pitches, and can be sent a primitive object like:

[HIT "quarter "metal "north-east "high-pitched]

This looks like a subroutine call (in prefix notation) with HIT being the subroutine's name and some arguments that are constants here (shown by the

Logo quote character (") which means: Take the next thing as a literal, do not handle it as a program). This example shows the use of one of the primitives we made for the instrument system controlling Ringo: a large percussion installation by Floris van Manen and Trimpin of the Klankschap Foundation (Manen and Trimpin 1986). The only other primitive needed was a rest:

```
[REST "whole]
```

For other instrument systems, different primitives would have to be designed. To give some examples:

```
[PHONEME "short "ooaAAH "low-
  pitched "very-loud]
[NOTE-ICON "eighth "G# "staccato]
[TAPE-RECORDER "A-77 "record
  (dB -10)]
[SINE (ms 100) (Hz 440)]
```

The first being appropriate for a phoneme synthesizer, the second for a score printing program, the third for a studio remote control system and the last for a digitally-controlled sine oscillator. Notice again the absence of arbitrary coding in the parameters. Scaling of parameters and decoding can be done in the primitives themselves as in the first example, or in a Logo function call, as in the last two, whichever is more appropriate. The primitive musical objects are not defined in the score but rather in the instrument part of the system. The only assumptions we make about primitives is their behavior with respect to time.

They are stretched out in time; but there is no specification in the primitives themselves as to when they have to take place. The start time is not a parameter of the basic musical objects. So we consider a sound now, and the same sound some time later as being the same musical object. The time-extent of a primitive is, however, predefined, and is computable from its call.

Time Structuring

We state that a score is a set of musical objects plus their timing relations. Specifying these relations is the task of our structuring functions. Only two

timing relations are needed. We must be able to express that musical objects take place at the same time, and we must be able to order them one after another. We call the first basic time order **PARALLEL** and the second **SEQUENTIAL**. A score can now be expressed in a nested structure of parallel and sequential musical objects. To give some examples,

```
[SEQUENTIAL theme improvisation theme
  coda]
```

means that there are musical objects called **theme**, **improvisation**, and **coda** combined in a sequence to make up a piece. We can make a small Logo program to represent this newly constructed musical object and give it a name:

```
TO jam
  OUTPUT [SEQUENTIAL theme improvisation
    theme coda]
END
```

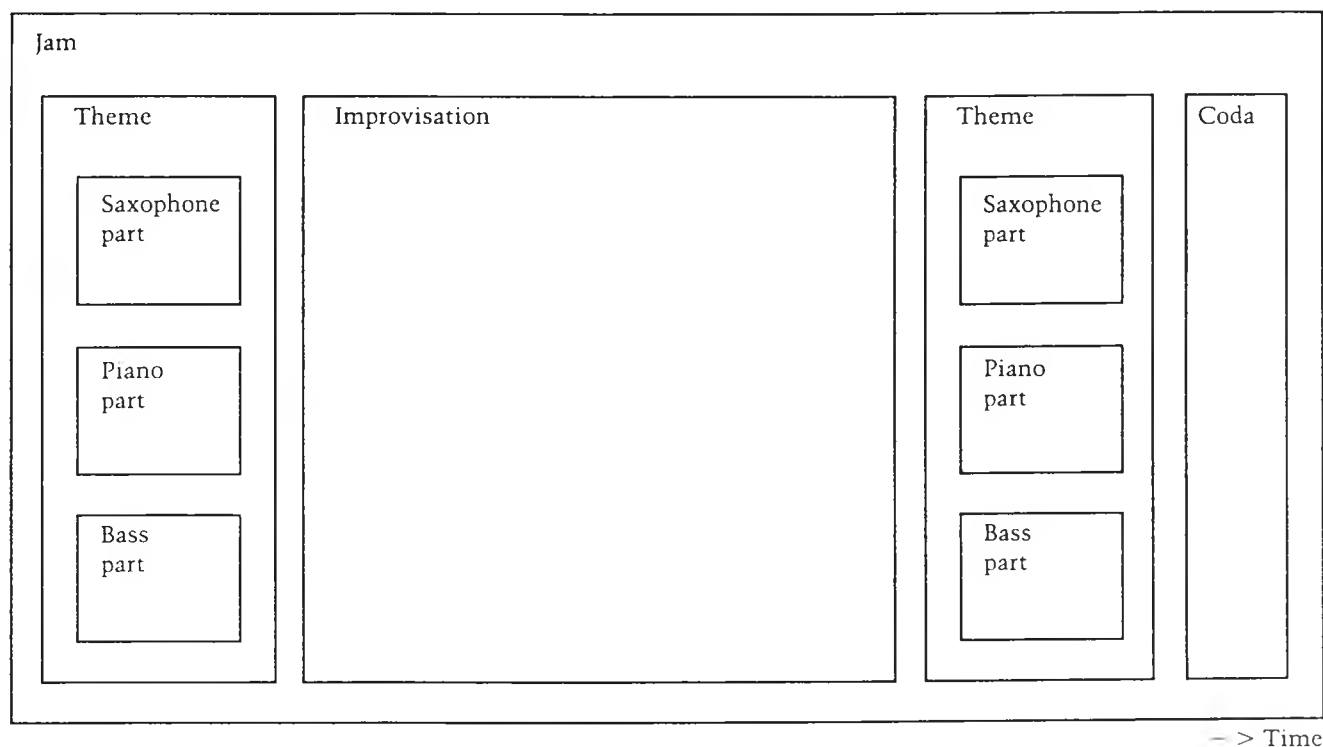
In which **TO** binds the program name **jam** to the program that, when run, will output the musical object. Now in its turn we can define **theme** as a layered musical structure:

```
TO theme
  OUTPUT [PARALLEL bass-part piano-
    part saxophone-part]
END
```

Which defines **theme** to have a **bass-part**, a **piano-part**, and a **saxophone-part** all starting at the same time. The parts can be defined as sequences of chords, notes, and rests. Chords can be described as parallel notes, etc. If these notes are primitive objects (are known by the instrument) we can stop here. A graphical representation of the defined object **jam** is shown in Fig. 1.

In the score system, **SEQUENTIAL** and **PARALLEL** can be looked upon as functions, as programs that create a time structure. But they can also be seen as data, representing the timing relations between several parts. This two sided approach, representing knowledge as a program and as data, gives the best of both worlds (the procedural and the de-

Fig. 1. Time structure of the object *Jam*.



clarative one). Only in Lisp-like languages this is possible. The process scheduling mechanism of FORMES (Rodet and Cointe 1984) uses the same time-structure functions, but they cannot be looked upon as data, so transformations on these structures are impossible. To summarize the score language properties:

Primitives are not given.

Time structures are made by nesting **PARALLEL** and **SEQUENTIAL** constellations.

Modularity and naming is achieved by the Logo function definition mechanism.

Advanced Timing Control

As an illustration of the power of not using absolute start times, we show two examples of advanced timing control. The first one, called **PRE**, shifts the start time of its argument. The "magical" musical object produced, will already be finished when asked to start. This is useful for constructing things like grace notes, upbeat, and the start of a tape re-

corder some seconds before it has to record. This mechanism prevents the cluttering of higher-level descriptions with low-level details. The second function, called **POST**, shifts the end time of a musical object to its start time (see Fig. 2). The object produced behaves like a sort of secret tail to the object it clings to, always coming after it but invisible to the outside timing. In this way grouping objects can be done on logical or musical grounds. Modifying **PRE** or **POST** objects does not change the general timing. It should be noted that the functions **PRE** and **POST** are polymorphic: they can be applied to objects on any level of our hierarchical score representation.

The Composer

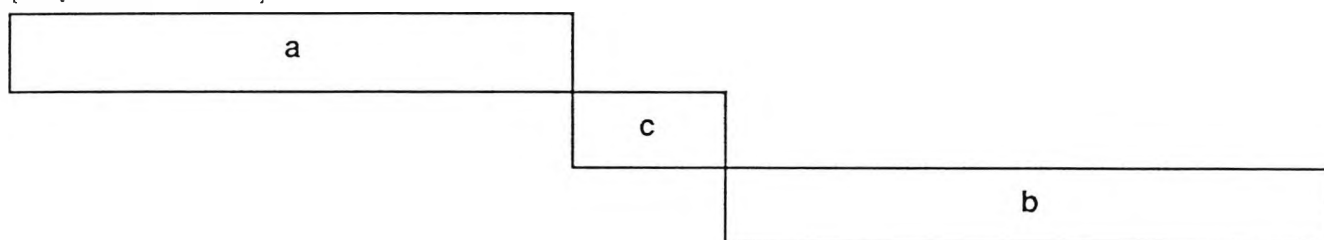
A composition system should not reflect implicit views of composition. We do not want to enforce compositional ideas on users of the system. The system should give access to different techniques and styles. We avoided making choices about "the

Fig. 2. The use of PRE and POST primitives.

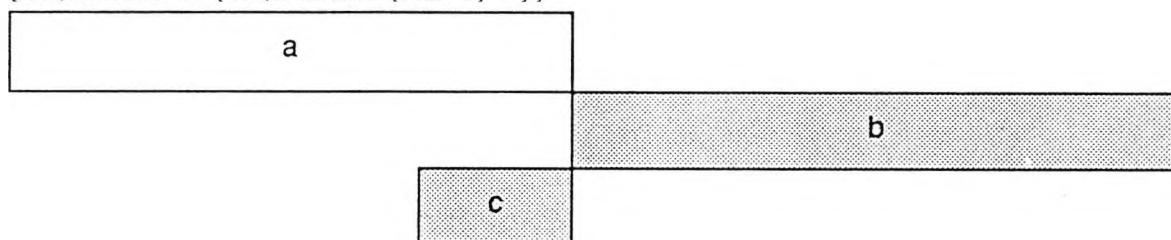
[SEQUENTIAL a b]



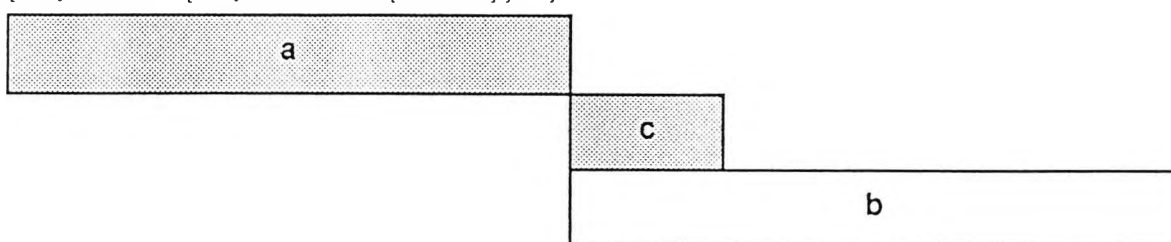
[SEQUENTIAL a c b]



[SEQUENTIAL a [SEQUENTIAL [PRE c] b]]



[SEQUENTIAL [SEQUENTIAL a [POST c]] b]



best way of composing," but instead made available a range of compositional techniques that are otherwise only available in different programs and institutions. Users can see for themselves, experiment with, and eventually make their own choices.

Because the ways of composing are myriad, and handling them all at once is very confusing, we selected a number of broad fields and covered them each in a separate package. We constructed each of these packages as a kind of complete workbench on

which a compositional approach can be experimented with. In the Logo community these packages are called *microworlds*. A microworld is a small set of powerful tools that constitute a more or less complete, or closed system (so there is no need of more or other tools within this world). This enables the user to learn about and make use of the knowledge domain expressed in these tools. The following paragraphs describe one of these music composition worlds.

"Composing is Making Choices"

Choices can be made by the computer or by the composer. This microworld focusses on the range of control a composer wants to have over the musical material. It enables the composer to prescribe the rules of possible choices to the computer. The computer makes the actual choices according to the rules. **LOCO** remains silent about the musical value of the choices; this is for the composer and the listener to decide.

Discrete Choices

In this section we focus on choices from a finite set of possibilities. Discrete or quantized choices reflect the concept of scales (be it in loudness, pitch, or duration) in which not all values are permitted. Our first-choice principle describes total control by the composer. All subsequent interpretations of the piece will be the same. This choice mechanism is called **CONSTANT**.

```
CONSTANT "instrument "sitar
```

generates a program called **instrument**. When it is run it outputs the word **sitar**. Our next choice principle is a simple aleatoric choice—as often used by different composers.

```
ALEATORIC "timbre [hollow fat  
rumbling ringing]
```

The program generator **ALEATORIC** is called with two arguments. The first argument is the name of the program to be created and the second is the list of possibilities (the event space) of this choice. After the program named **timbre** is created, it can be called anywhere in a Logo program and produces a random timbre. Of course, the Logo interpreter can be used directly to check its workings.

```
?ALEATORIC "timbre [hollow fat  
rumbling ringing]  
?timbre  
fat  
?timbre  
ringing
```

(Note that the computer output is printed light, and the question mark is the Logo prompt.)

Neither **ALEATORIC** itself, nor the program **timbre** knows about the actual timbres mentioned; they just handle words. These words only receive an interpretation at a later stage. In this way it is possible to make use of the same choice mechanisms for different types of musical objects.

```
ALEATORIC "piece [part.1 part.2  
part.3 part.4 part.5]
```

This realizes parts chosen at random. Using aleatoric choices for the pitches in a melody yields a so-called *white melody*. It lacks, like in white noise, any structure or predictability. A natural extension to **ALEATORIC** is the possibility of assigning different probabilities to the elements of its event space, like loaded dice. The generator **WEIGHTED** provides such a feature. It has the same argument structure as **ALEATORIC** but the elements of the event space are paired (in a list) with the probability that they will be chosen. Probabilities are specified as real numbers between 0 and 1.

```
WEIGHTED "instrument [[snaredrum 0.5]  
[timpani 0.2]  
[cymbal 0.3]]
```

When **instrument** is called, 50 percent of the time it outputs **snaredrum**, 20 percent of the time it outputs **timpani**, and the rest of the time it will produce **cymbal**.

Sometimes there is a large number of numeric elements to choose from. Writing them all in a list for **ALEATORIC** would be too cumbersome. For this we can use the generator **SCALED** instead.

```
SCALED "pulse 1/4 2 1/8
```

The program **pulse** will return values between 1/4 and 2, in a scale with a resolution (grid) of 1/8.

The preceding choices are made without a memory. That is, a choice does not depend on the outcome of previous ones. We now introduce choice processes that have a memory—an internal state. Instead of the term *stochastic variable* to refer to

Fig. 3. Rhythmic structure
resulting from the program
Duration.



the outcome of a choice, we have to use the term *stochastic process* now. Our first most well-known stochastic process is called **SERIAL**. Each time a choice is made the event space is reduced by the chosen element, so that in a next choice this element is excluded. Once the event space has been emptied, it is reestablished to its initial value. This constitutes a generalization of the twelve-tone principle. For example:

```
SERIAL "duration [eighth quarter
    sixteenth sixteenth]
```

A rhythmic structure produced by this procedure is shown in Fig. 3. Sometimes the composer wants to have complete control over a time ordering. In that case **ORDERED** can be used. The elements of its event space appears in the predefined order.

```
ORDERED "accent [heavy none light
    none]
```

This produces **heavy none light none heavy none light . . . ad infinitum**. There are different ways in which primitives, choices, and time orderings can be combined. For example, the arguments of primitives can be the result of choice programs.

```
SERIAL "duration [whole half quarter
    quarter]
ALEATORIC "pitch [c d f g a]
CONSTANT "loudness "pp
```

If we had a primitive musical object **NOTE** that takes a duration, a pitch, and a loudness as arguments, then:

```
[NOTE duration pitch loudness]
```

results in a note of a random pitch (in a pentatonic scale) and a random duration (with a serial structure) in a constant pianissimo.

Choices can also be embedded in choices as in the next example:

```
ORDERED "element
    [[NOTE duration pitch loudness]
    [REST duration]]
```

This produces an alternating sequence of the previously mentioned notes and rests.

Also time orderings can be embedded in choices and vice versa:

```
ALEATORIC "structure
    [[PARALLEL element element]
    [SEQUENTIAL element element]]
```

The results of choices can be used in calculations like the following:

```
SERIAL "error [small-positive zero
    small-negative]
```

Adding **error** to the **duration**, of a musical object, constitutes a first experiment in forming a *rubato*. For further hierarchical nesting of choice principles, we need a "dereferencing" mechanism, which we call **EVALUATED**. Let us start with three value generators, **high**, **mid**, and **low**.

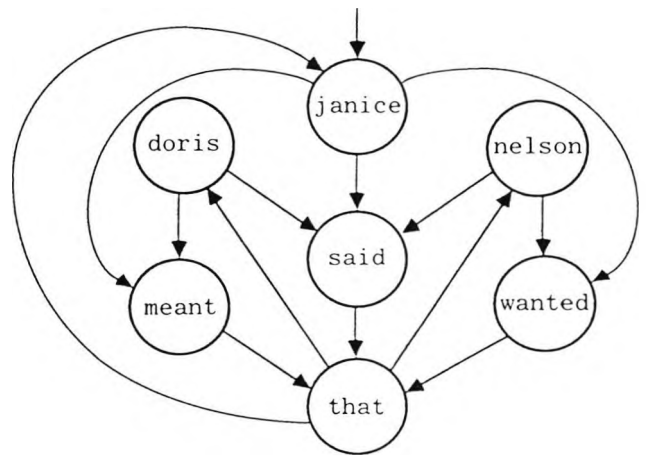
```
ALEATORIC "high [400 500 600 700]
ALEATORIC "mid [200 250 300 350]
ALEATORIC "low [100 125 150 175]
```

When we want to choose the generator to use for the choice in a fixed order, we can construct the expressions:

```
ORDERED "register [high mid low]
EVALUATED "pitch "register
```

Register will just produce the words **high**, **mid**, and **low**, in that order. **Pitch** produces the high, middle, low, and frequency value itself.

Fig. 4. Transition network of the object Word.



A different way to produce the same result is to make a program that generates a list of possible values at a time, and to use that program as an argument for ALEATORIC:

```
ORDERED "possibilities
[[400 500 600 700]
 [200 250 300 350]
 [100 125 150 175]]
ALEATORIC "pitch "possibilities
```

Each time that *pitch* is called, it first evaluates possibilities and then picks a random value from the list that was produced by possibilities.

Linking choice principles in a network, each one choosing a next one to use, can be done by a generator called TRANSITIVE. Like all generators, it has the name of the program to be generated as its first argument. The second argument is a starting state. Besides TRANSITIVE we need the definitions of some choice programs that produce the name of other choice programs.

```
TRANSITIVE "word "janice
ALEATORIC "janice [meant said wanted]
ALEATORIC "doris [meant said]
ALEATORIC "nelson [said wanted]
ALEATORIC "meant [that]
ALEATORIC "said [that]
ALEATORIC "wanted [that]
ALEATORIC "that [doris nelson janice]
```

Each time *word* is called, it returns its current state, starting with *janice*, and uses that state as a choice principle to calculate its next state. So calling *word* several times produces something like *janice said that nelson wanted that janice meant that. . .* The flow of control implemented by the programs is called a *transition network*. It looks familiar in its graphical representation (Fig. 4).

By using WEIGHTED instead of ALEATORIC as the basis of a transition network, the well-known *Markov chains* can be modelled. Probabilities are then not assigned to the event space but to the transitions from one event to the next. This powerful tool, which is able to model all discrete stochastic

processes, can be depicted by labelling the arcs of a transition network with probabilities. Our way of constructing Markov chains has the added advantage of dividing data in manageable pieces, instead of using huge matrices. Furthermore, by incorporating other choices (like SERIAL or ORDERED) in a transition network, many new musical experiments are possible.

A way of constructing a stochastic process from a simple stochastic variable is to accumulate (or integrate) its previous values. CUMULATIVE is a program generator that defines a program that accomplishes this.

```
ALEATORIC "interval [1 2 3 -6]
CUMULATIVE "pitch "interval 60
```

In this way a program named *pitch* is generated that outputs 60 when it is called for the first time. Each subsequent time it is used, it yields a value produced by adding 1, 2, 3, or -6 to its previous value. Such series of pitches fall into the class of *brownian melodies* in which a new state is calculated by incrementing a present state with a small value. Brown melodies can be compared to white ones and are easily distinguished from them by ear. Different kinds of these random walk algorithms can be experimented with by changing the input of CUMULATIVE (the *interval* choice mechanism). CUMULATIVE can also be used to produce values that change with a constant linear slope.

Fig. 5. Pitch changes produced by a Voss algorithm.

CUMULATIVE "count.down -1 10

Count.down indeed counts down from 10, one at a time.

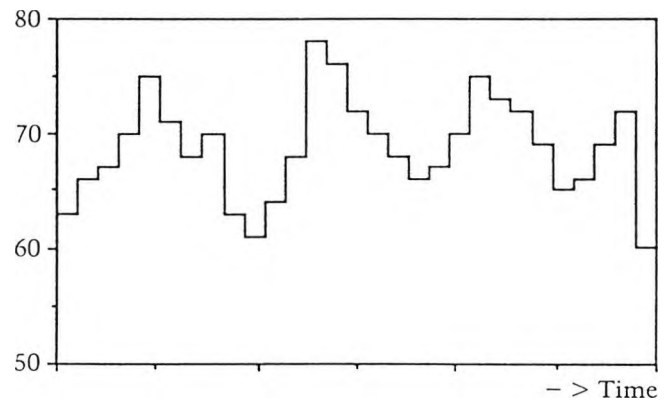
Since duplication of values (using the same value for a second time) is encountered so often, we constructed a generator ITERATIVE that can handle this mechanism.

ITERATIVE "repeated.value "value 3

If the user has defined or generated a program named value (of any kind), the program repeated.value yields the same subsequent results as this program, but now each value will be repeated three times. Repeated.value can return something like low, low, low, high, high, high, mid, mid, mid. . . . This mechanism gives us a simple way to construct a program for generating 1/f melodies (Voss and Clarke 1978).

SCALED "fast.value 20 26 1
ITERATIVE "mid.value "fast.value 2
ITERATIVE "slow.value "fast.value 4

While fast.value yields a new number between 20 and 26 each new round, mid.value produces a new value in the same range only once in two rounds, and slow.value comes up with something new every four rounds. We add the three produced values and use fast.value + mid.value + slow.value as a pitch number. Once in a while there can be a big jump in pitch because all three values change in the same direction. When there are two values changing, which happens more often, the jump is smaller. Most of the time there will be only a small jump because only one value is changing (see Fig. 5). The resulting pitch pattern is a raw approximation of a 1/f melody. The possibilities for modification and refinement are enormous. The values can be scaled properly to give the higher-frequency components less amplitude. The repeat values 2 and 4 could be calculated—or used elsewhere to produce a rhythmic structure related to the melody. The aleatoric choice could be replaced by another process and so forth. Experimenting with these melodies, comparing them to brown and white ones, will be made



much easier by using our open and simple implementation instead of awkward Basic programs (Dodge and Bahn 1986).

Formal languages and grammars have received attention in music research. They were used as a model to describe different musical domains or as a mechanism for new (computer) compositions. (For an excellent review of their possibilities and use, see [Roads 1985].) Just as choice principles can be built on other ones, they can also refer to themselves [Desain and Honing 1988].

ALEATORIC "answer [low.note
[SEQUENTIAL note answer]]

Answer results in a low note or a note followed by an answer. The last answer produces a low note or a note followed by an answer. In this way, when the system is asked to interpret an answer, it plays a sequence of notes ending with a low note. To expand our grammar:

ALEATORIC "dialog [nothing
[SEQUENTIAL question answer dialog]]
ALEATORIC "question [high.note
[SEQUENTIAL note answer]]
ALEATORIC "answer [low.note
[SEQUENTIAL note answer]]

This context-free grammar generates series of question-answer pairs. Of course, the programs note, high.note, low.note, and nothing have to be supplied. These programs function as terminals

Changing the choice principle used to choose the grammar rule, from **ALEATORIC** to **WEIGHTED**, will yield a so-called *programmed grammar*. The individual probabilities assigned to the rules can be used to control the average size, and amount of appearance of the produced substructures. Assigning a high probability to production rules containing only terminals produces small objects and so forth.

```

ALEATORIC "note
[nothing
  [SEQUENTIAL quarter note quarter]
  [SEQUENTIAL eighth note eighth]
  [SEQUENTIAL sixteenth note
sixteenth]
  [SEQUENTIAL note quarter.rest note]]

```

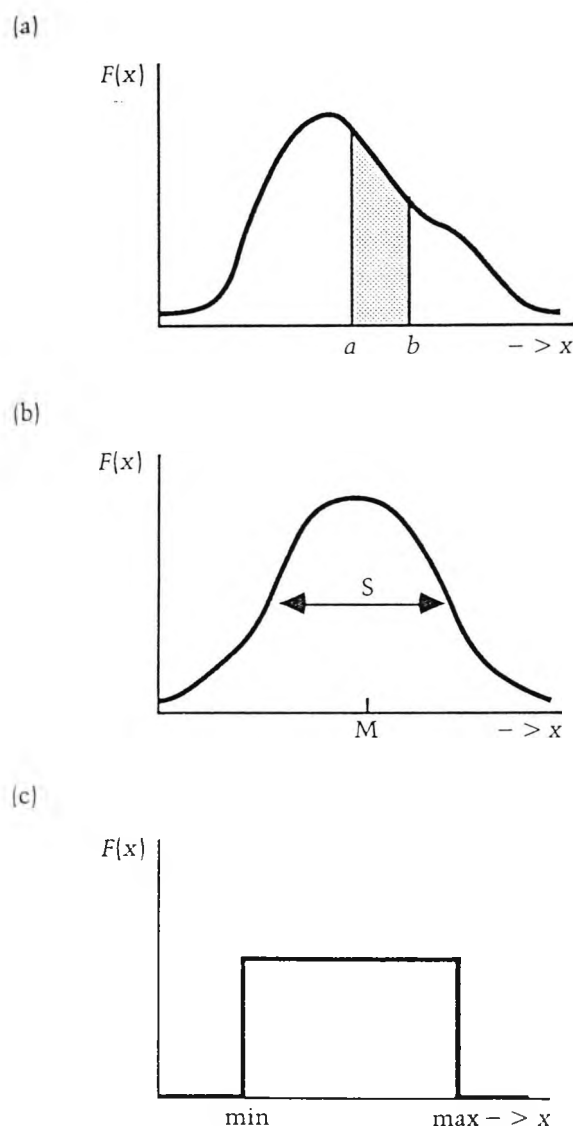
Grammars will produce highly structured music, and are most useful in describing such music. An example of a grammar for chord progression in traditional twelve-bar blues is described by (Steedman 1984). Because the grammar capability is implied by the total structure of the LOCO system, it can be intermingled in various ways with the other mechanisms. Note the restriction to context-free grammars.

Traditional scales have been overused, so that for some composers they represent a restriction to be eliminated. In electronic and computer music one can specify a continuum of frequencies in Hertz or cents, and a continuum of loudness in decibels, for

The main tool for describing stochastic variables is the *probability density function*. It relates an interval to the chance that a variable will be in this interval. In Fig. 7(a) an arbitrary probability density function $F(x)$ is drawn. The probability that x is between a and b is the (shaded) area below F between a and b . This means that all probability density functions will be nonnegative functions with a total surface area of 1. A commonly used probability density is the *normal* or *Gaussian distribution* (Fig. 7(b)). It is characterized by a mean value M (the position of the peak in the function), and a deviation value S that signifies how wide the peak is. There is also the *uniform distribution* (Fig. 7(c)) in which any value between a specified minimum and maximum has an equal chance to occur. For our microworld we implemented these distributions as program generators called **GAUSSIAN** and **UNIFORM**. They can create a program that picks a value according to its distribution.

When the program **frequency.a** is called (e.g., typed into the Logo interpreter), it returns a value. The probability density of this value will be the normal distribution with mean of 1000 Hz and deviation of 200. The program **frequency.b** also returns values around 1 KHz. But they are more in the neighborhood of 1000 compared to the values produced by **frequency.a**, owing to its smaller deviation. The generated program **loudness** produces values between -80 and 0. There is no neighborhood in which they are likely to fall (except of course the interval from -80 to 0 itself).

Fig. 7. Continuous distributions. (a) Arbitrary probability function. (b) Normal or Gaussian distribution. (c) Uniform distribution.



Frequency.a and frequency.b are mutually independent: knowing the value produced by one does not help in estimating the unknown value produced by the other. Sometimes we want to describe two stochastic variables that are not mutually independent. Perhaps we want to create random pitches on two instruments, which must have a tendency of one pitch being high, if the other one is also high. Then there is a need for a *two-dimensional distribution function*. We'll just describe one.

The two dimensional normal or Gaussian distribution has five parameters. A mean value for the first variable, and a mean for the second. Also two deviations are needed, one for each variable. The last parameter, called the *covariance*, describes how much both variables are correlated. If the covariance is high, an educated guess can be made at the second variable when the first is known. The generated programs will return a list of two values that are drawn according to the parameters.

```
GAUSSIAN. 2D "independent 0 1 0 1 0
GAUSSIAN. 2D "small.corr. 0 1 0 1 .3
GAUSSIAN. 2D "large.corr. 0 1 0 1 .9
```

The plots in Fig. 8 were made by plotting these two values as x and y coordinates. It is simple to generalize this process to other distribution functions and more dimensions. The continuous distributions can be combined in various ways with other objects. We can, for example, vary the parameters of a distribution.

```
ALEATORIC "mean [1 2 5]
GAUSSIAN "near "mean 1
```

The resulting distribution of **near** is shown in Fig. 9. Each time the program **near** is run, it first draws a mean value. This value is used in the normal distribution of **near**. So it results in a value around 1, 2, or 5. This stacking of random choices yields, in a conceptually simple way, stochastic variables with sophisticated probability densities.

The same method can be used to construct *tendency masks* in only a few lines.

```
CUMULATIVE "min -10 1
CUMULATIVE "max 20 -2
UNIFORM "value "min "max
```

The program **value** produces values from a permitted range, this range extends from -10 to 20, gradually diminishing (Fig. 10). Combining the proposed building blocks in slightly different ways gives us an enormous field of possibilities (tendency masks with internal time-changing distributions, distributions with time-changing parameters, etc.).

Fig. 8. The use of a two-dimensional probability distribution with different correlations.

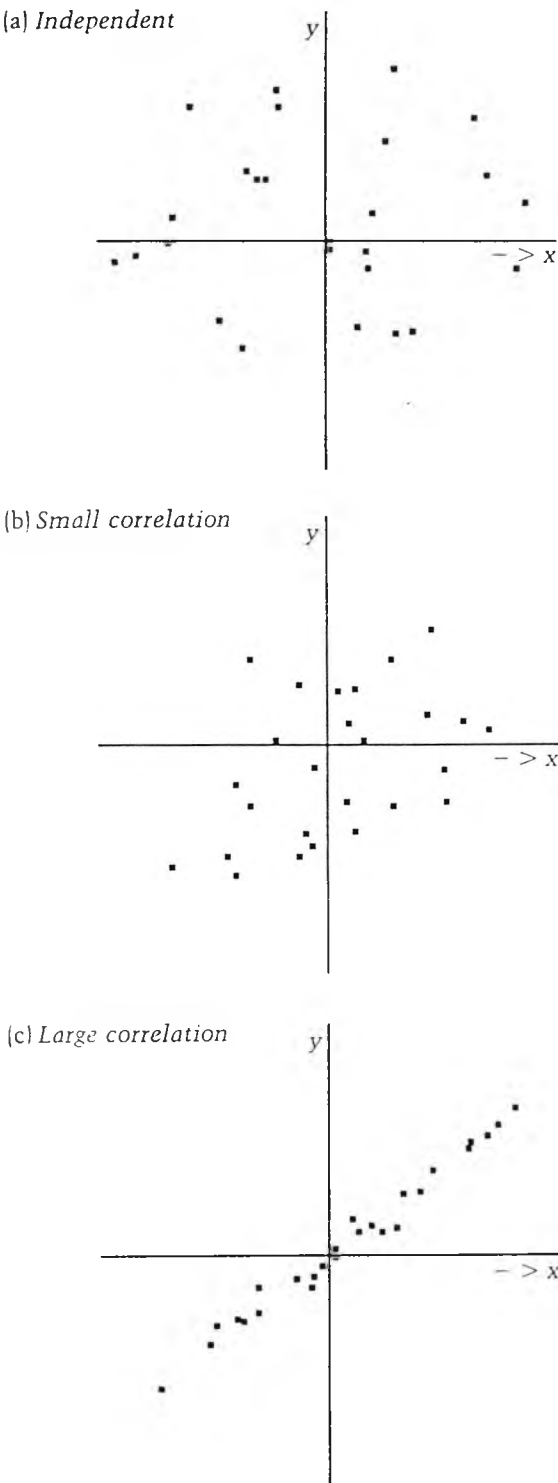
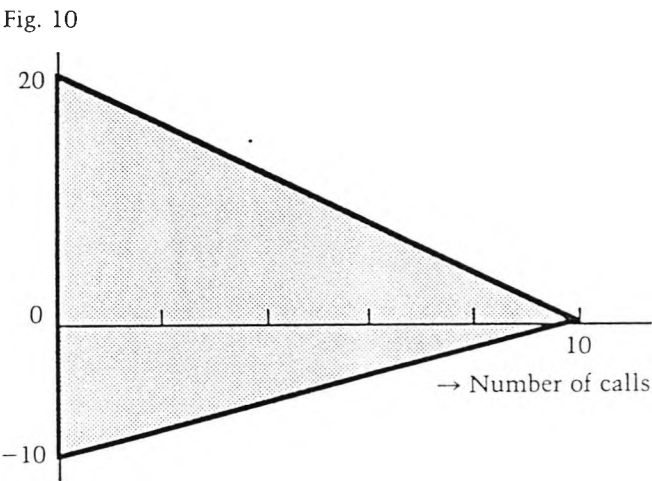
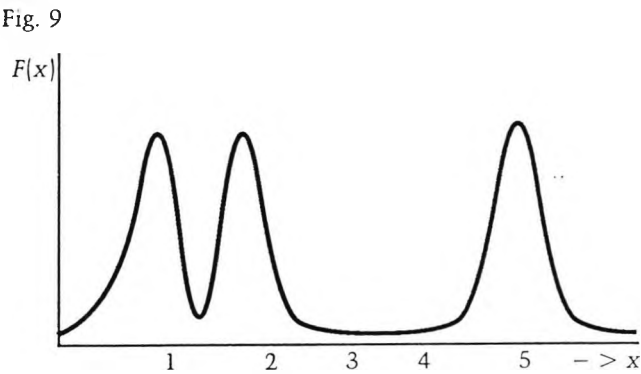


Fig. 9. Distribution of Near.



If we had adhered to the traditional way of defining mechanisms, we would have surely defined one sort of tendency mask, thus forcing the user to use our concept of it. We hope we have proved that making small but powerful mechanisms that can be combined in any thinkable way is a better approach.

Conclusion

We have used LOCO in a number of courses and workshops. It has proven to be a rich, motivational context for different kinds of participants. After a short explanation they were able to start using LOCO and soon were expressing their own ideas, depending on previous knowledge and experience in the field of traditional or computer music. The sys-

tem is currently used at the music faculty of the recently founded Center for Art, Media and Technology (CKMT) for courses in computer music composition.

We are still continuing our search for more mechanisms used in different (computer) music styles, and their expression in LOCO microworlds. Current work on composition systems is done in Common Lisp, so LOCO becomes COCO (Desain 1988). The research is directed towards transformations, pattern matching, parallelism, graphical interfaces (Desain 1986) and real-time interaction.

Implementations

We have available an implementation for the Apple IIe with Terrapin Logo V3.0, one for the Macintosh and Macintosh II with MicroSoft/LCSI Logo, and a Dutch version for the Yamaha CX5M-II with Philips/LCSI MSX-Logo. The cost of a disk and a manual is \$50 (postage and handling included). Checks should be made payable to the Center for Art, Media and Technology, Utrecht Academy of Arts, in American dollars drawn on a Dutch bank.

Acknowledgments

We would like to thank Janet Cornish, Floris van Manen, Martin Seeley, and Marjon de Kleine for their valuable comments on earlier versions of this paper and all our colleagues at CKMT for support and facilities.

References

- Anderson, J., A. Corbett, and B. Reiser. 1987. *Essential Lisp*. Reading, Mass.: Addison-Wesley.
- Beckwith, S. 1975. "The Interactive Music Project at York University." Available on microfiche from ONTERIS, Ministry of Education, Toronto.
- Berg, P. 1979. "PILE—A Language for Sound Synthesis." *Computer Music Journal* 3(1):30–41. Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music*. Cambridge, Mass.: MIT Press, pp. 160–190.
- Desain, P. 1986. "Graphical Programming in Computer Music, A Proposal." In P. Berg, ed. *Proceedings of the 1986 International Computer Music Conference*. San Francisco: Computer Music Association, pp. 161–166.
- Desain, P. 1988. "LISP as a Second Language, Functional Aspects." Submitted to the International Computer Music Conference, 1988.
- Desain, P., and H. Honing. 1988. "LOCO: A Composition Microworld in Logo." *Computer Music Journal* 12(3).
- Dodge, G., and C. Bahn. 1986. "Musical Fractals." *Byte* (6):185–196.
- Fry, C. 1984. "Flavors Band: A Language for Specifying Musical Style." *Computer Music Journal* 8(4):20–34.
- Harvey, B. 1985. *Intermediate Programming. Computer Science Logo Style, Vol. 1*. Cambridge, Mass.: MIT Press.
- Jones, K. 1981. "Compositional Applications of Stochastic Processes." *Computer Music Journal* 5(2):45–61.
- Koenig, G. M. 1971. *Project Two*. Utrecht: Institute of Sonology.
- Manen, F. van, and Trimpin. 1986. "Ringo: A Percussive Installation." In P. Berg, ed. *Proceedings of the 1986 International Computer Music Conference*. San Francisco: Computer Music Association, pp. 193–196.
- Roads, C. 1985. "Grammars as Representations for Music." In C. Roads and J. Strawn, eds. *Foundations of Computer Music*. Cambridge, Mass.: MIT Press, pp. 403–442.
- Rodet, X., and P. Cointe. 1984. "FORMES: Composition and Scheduling of Processes." *Computer Music Journal* 8(3):32–48.
- Schottstaedt, W. 1983. "PLA: A Composer's Idea of a Language." *Computer Music Journal* 7(1):11–20.
- Steedman, M. J. 1984. "A Generative Grammar for Jazz Chord Sequences." *Music Perception* 2(1):53–77.
- Voss, R. F., and J. Clarke. 1978. "Music from 1/f Noise." *Journal of the Acoustical Society of America* 63(1):258–263.
- Xenakis, I. 1971. *Formalized Music*. Bloomington: Indiana University Press.

LOCO MANUAL

1 INTRODUCTION

LOCO is a system for (music) composition. It is built on the general programming language Logo. It can be used as a workbench for experimenting with musical structures, both for beginners and professionals. It is a non-realtime system, which means that composing and playing don't take place at the same time. **LOCO** consists of three subsystems: the composition system, the score system, and the instrument system (see figure 3). The composition system consists of a set of program generators, programs that write programs. They can be used to create the so-called choice systems that will make a composition according to the rules provided by the composer. The connection between the composition system and the instrument is via the score system. The instrument system uses standard MIDI codes to control any kind of MIDI synthesiser.

The scope of this manual is a step by step introduction to **LOCO**. It is not intended as a demonstration of the full power of **LOCO**, nor to explain theories about music composition. The underlying philosophy and design decisions, as well as some other examples of **LOCO** are given elsewhere (Desain/Honing 1986,1988).

2 GETTING STARTED THE VERY FIRST TIME

This chapter explains the configuration actions needed to start **LOCO** for the very first time at your site. For most users, this will already have been done by someone else, so they can skip this chapter and start reading chapter 3.

2.1 Needed Equipment

1. An Apple Macintosh computer: Macintosh 512K, Ed, Plus or II.
2. MIDI interface (i.e. Apple, Applica, Opcode)
3. MIDI cable.
4. MIDI synthesizer - any kind of MIDI synthesiser will do, even a synth without a keyboard can be used (like the Yamaha TX-7 or FB-01 expander). Other MIDI driven equipment (like drummachines) can also be used.
5. Audio equipment - a simple headset will do for most experiments.
6. Microsoft/LCSI Logo
7. the **LOCO** disk.
8. Empty disks - for storing your compositional work and scores.

Note: **LOCO** runs only under the Finder (Don't use the Multifinder).

For workshops and classes in computer music composition it is often wise to concentrate first on **LOCO** itself, not being distracted by all kinds of fascinating audio equipment. For the same reason we advise to use synth's without a keyboard.

2.2 Step 1: Make a Working Copy of Your LOCO Disk

Make a copy of the LOCO disk in the normal way (See your Macintosh manual). Store the original disk safely and use the working-copy for the configuration process.

Making copies of LOCO for other than backup purposes is prohibited. They can be traced back to the original purchaser who will be held responsible for all.

2.3 Step 2: Connecting a MIDI Interface and Synthesiser

Connect the MIDI interface to the modem port. For primitives adjusting the default settings see 6.8.

Now connect the MIDI-out of the interface to the MIDI-in on your synthesiser with a standard MIDI cable. Make sure the synthesiser can receive MIDI information on channel 1 (see your synthesiser manual). Additional synthesisers may be "daisy chained". They can receive MIDI on subsequent channel numbers.

2.4 Step 3: Configuring Your LOGO

Make a new copy of your Microsoft Logo application and call it LOCO Logo (see page 204 of the Microsoft Reference Manual).

After this startup the Preferences Program on your Microsoft disk. (If you use a Macintosh with HFS, make sure both the Preferences Program and a copy of the Logo application are not inside a folder. The Preferences Program is affected by the HFS system and can not find primitive sets that are in folders, see Addendum Microsoft Manual).

Next, copy the LOCO primitive set called LOCOPSET.P into your LOCO Logo using the Primitive Set Mover (see page 214/217 of the Microsoft Reference Manual).

Finally, if you want to adjust your copy of Logo to your personal needs (i.e. more workspace) select Memory & Font Preferences in the Preferences Program to adjust the allocation of memory and default font (see page 207/210 of the Microsoft Reference Manual).

The configuration phase is now done. You can quit the Preference Program and copy LOCO Logo to your working copy of the LOCO disk. Place them in the same folder.

You can now continue with chapter 3.

3 GETTING STARTED

In this chapter it is assumed that your Macintosh is connected to a synthesizer, receiving MIDI data at channel 1, and the **LOCO** disk is assumed to be configured appropriately.

You can then startup **LOCO** by double-clicking on the file called "LOCO startup".

Feel free to stop reading the next chapters at any time and do some experiments of your own - doing is the best way of learning.

4 THE SCORE SYSTEM

After starting **LOCO**, a score writing and playing system is available. This score system is to be considered as a tool - it is in itself not so interesting (see figure 3). But with this tool you can make programs that write music on the score in their own (slow) way. After that the score can be played as fast as you like by your synthesiser.

On a score you can write simple and compound musical objects. An example of a simple musical object is a **NOTE**:

NOTE

```
[NOTE duration pitch loudness]
```

It has 3 inputs: a duration, a pitch and a loudness. The duration signifies how long the note is going to last. The pitch is a number counting the semi-tone steps of an equal-tempered scale, middle C has pitch number 60 (see chapter 6.7 for a table of MIDI pitches). The loudness is a number between 0 and 1. Later we will define translations from well-known names (like middle C and pianissimo) to these numbers. Let's first write one simple short loud middle C note to the score and play it.

**WRITE
PLAY**

```
WRITE [NOTE 1 60 1]  
PLAY
```

(In all examples plain text is user input and italic is computer output.)

In Microsoft Logo there is no prompt. You can type text everywhere in a text window. To let the interpreter actually do the instruction, press the Enter key after each command line (see page 11/12 of the Microsoft Reference Manual).

If we write more notes they will be added one after another to the score. Let's try some other inputs:

```
WRITE [NOTE 2 60 1]  
WRITE [NOTE 1 58 1]  
PLAY
```

REST

Of course we also need rests:

```
WRITE [REST 1]  
WRITE [NOTE 2 58 0.7]  
PLAY
```

When you make a typing error you can use the common Logo cursor keys and line editing commands (see page 9 of the Microsoft Reference Manual).

Clover-S stops the playing of the score and clover-H for Help information also works for **LOCO** primitives.

Also other Logo constructs like repetition are usable within **LOCO**:

```
REPEAT 4 [WRITE [NOTE 1 67 1]]  
PLAY
```

If you are bored with such repeated notes or with typing all these notes yourself, and want to let the computer compose the piece for you, you should start reading chapter 5 and come back later to hear all about the facilities of the score system. On the other hand, if you like to explore one thing at a time just continue reading.

POSITION
POSITION?

Let's talk about polyphony now. We can start from the beginning of the score (POSITION 0) and add a second layer of notes. Or we can see where we are in the score, go to a specific position, and continue from there.

```
POSITION 0
WRITE [NOTE 3 51 1]
WRITE [NOTE 1 52 1]
PLAY
POSITION?
4
POSITION 2
WRITE [NOTE 1 50 1]
PLAY
```

The created score in a pianoroll notation looks like this:

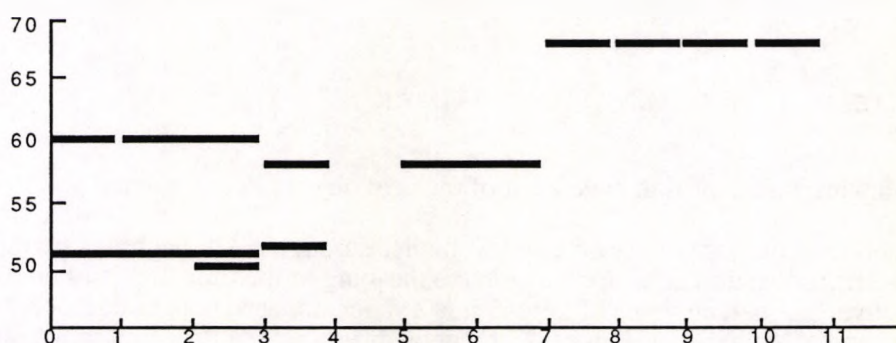


Figure 1. Pianoroll notation of the created score.

The score can be played at different tempos. A tempo number gives the number of time-units per minute. So our little score of 11 time-units will last for 11 seconds when played at tempo 60:

TEMPO
TEMPO?

```
TEMPO?
60
PLAY
TEMPO 120
PLAY
TEMPO 60
```

SAVE.SCORE
ERASE.SCORE

A score can be saved to disk, erased and retrieved at a later time. **SAVE.SCORE** will open a Save dialog box and asks for a filename.

```
SAVE.SCORE
ERASE.SCORE
PLAY
WRITE [NOTE 1 60 1]
PLAY
LOAD.SCORE
PLAY
```

LOAD.SCORE

LOAD.SCORE overwrites the current score. So if you want to keep it, save it before reading a new score from disk. **LOAD.SCORE** opens a Load dialog box and lets you select the file that has to be loaded.

Next to make multiple layers of notes on the score by writing layer per layer and using **POSITION**, we can make compound musical objects at once (note that later these objects will be calculated by some kind of program, but for the moment we will type them in). Musical objects that will sound at the same time are constructed using **P** for Parallel. A parallel object will last as long as its longest component.

P

```
ERASE.SCORE
WRITE [P [[NOTE 4 60 1][NOTE 8 63 1]]]
PLAY
WRITE [P [[NOTE 4 57 1][NOTE 4 59 1][NOTE 4 63 1]]]
PLAY
```

You can make a sequential ordering of musical objects into one compound object by using **S** for Sequence.

S

```
ERASE.SCORE
WRITE [S [[NOTE 1 60 1][REST 1][NOTE 1 61 1]]]
PLAY
```

P and **S** are functions that have a list of musical objects as argument.

Sometimes you want to use notes with smaller durations. Although it is permitted to use fractional durations, a change in the meaning of the time-unit is often more effective. The default value of time-unit is a sixteenth, each note of duration 1 will be a sixteenth note - lasting for 0.25 seconds. If you set the time-unit to a quarter note, and since at tempo 60 you will have 60 of them each minute, a note of duration 1 will last for one second. Changing the time-unit will only affect the writing of the score (way of notating) and not the actual playing. If this all seems too complicated to you, just remember the following. If your tempo has to be set at an unreasonable high value to get the result you want, then it is better to write the score with a smaller time-unit and vice-versa. **TIMEUNIT** is usually set before you write the score, **TEMPO** while playing it (see figure 3).

TIMEUNIT TIMEUNIT?

```
ERASE.SCORE
TIMEUNIT?
1 / 16
TIMEUNIT 1/4
WRITE [NOTE 1 60 1]
WRITE [NOTE 1 61 1]
PLAY
ERASE.SCORE
WRITE [NOTE 1 60 1]
WRITE [NOTE 1 61 1]
PLAY
```

Note that **LOAD.SCORE** and **ERASE.SCORE** reset **TIMEUNIT** to its default value 1/16. That is why in the example after **ERASE.SCORE** the written notes sounded shorter. **TIMEUNIT** was set to its default value.

Most MIDI synthesisers are equipped with different sounds called instruments, timbres or voices. The playing instrument can be changed using **INSTRUMENT**. It has a number as input. This is the same number you'll have to use when selecting the instrument manually on the synthesiser itself.

INSTRUMENT INSTRUMENT?

```
ERASE.SCORE
WRITE [NOTE 4 54 1]
INSTRUMENT 3
PLAY
INSTRUMENT?
3
INSTRUMENT 22
PLAY
```

PART PART?

If you have a synthesiser that can handle more instruments at once (this is called multi-timbral), or if you have more synthesisers connected in a "daisy chain", each receiving MIDI information on a different channel, you can write music that has more then one part. If you do not have this, skip the explanation of **PART**. **PART** changes the current part of the score you are writing (i.e. the MIDI channel the notes will go to when played). And **INSTRUMENT** changes the instrument playing the current part.

```
ERASE.SCORE
PART?
1
WRITE [NOTE 1 60 1]
WRITE [NOTE 2 63 1]
INSTRUMENT 3
PLAY
PART 2
POSITION 0
WRITE [NOTE 2 55 1]
WRITE [NOTE 1 56 1]
INSTRUMENT 11
PLAY
PART 1
INSTRUMENT 10
PLAY
```

The resulting score can be visualised as follows:

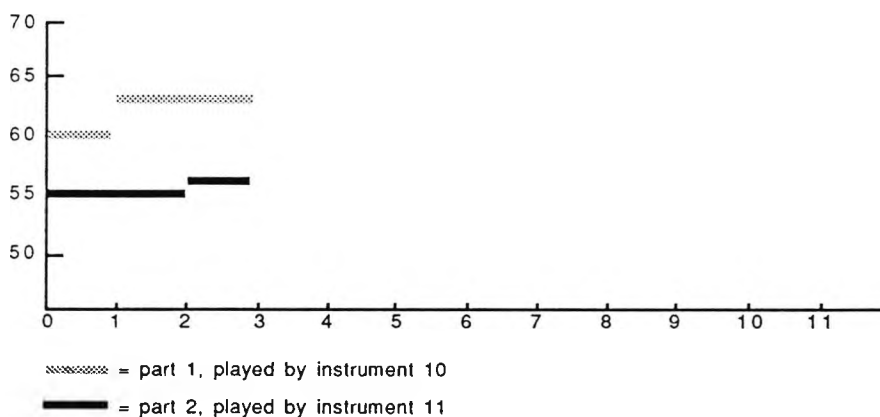


Figure 2. Pianoroll notation of the given score.

Note that **LOAD.SCORE** and **ERASE.SCORE** reset **PART** and **TIMEUNIT** to their default values, 1 and 1/16 respectively.

Advanced control of timing achievable with the **LOCO** system will be demonstrated next.

When writing scores the position in the score can be manipulated by the musical objects themselves. **[PRE object]** will be written to the score just before its current position without changing it. This is useful for objects like grace notes.

PRE

```
ERASE.SCORE
REPEAT 4 [WRITE [NOTE 1 60 1]]
PLAY
POSITION 3
WRITE [PRE [NOTE 0.2 50 1]]
PLAY
```

The **PRE** object is prefixed to the current position in the score, i.e. it already happened when it's asked to start at position 3 (you can hear that the **PRE** note is not on the same beat as the first layer).

PRE can also be combined in **S**.

```
POSITION 4
REPEAT 4 [WRITE [S [[PRE [NOTE 0.15 70 1]][NOTE 1 62 1]]]]
PLAY
POSITION?
8
```

PRE behaves like a prefix attached to **[NOTE 1 62 1]**, but it does not change the duration, nor the timing of the **S** object. The **POST** object is postfixed to the position in the score. This means that it can be written to the score at the current position without changing it. Mind that writing a **PRE** object at position 0 of the score is an error (This is like writing a musical object on your desk instead of on the music paper).

The next example will yield a chord:

POST

```
ERASE.SCORE
POSITION?
0
WRITE [POST [NOTE 1 70 1]]
POSITION?
0
WRITE [NOTE 4 60 1]
POSITION?
4
PLAY
WRITE [POST [NOTE 1 70 1]]
POSITION?
4
PLAY
```

To make clear the information flow in the score system we summarize the commands in a picture:

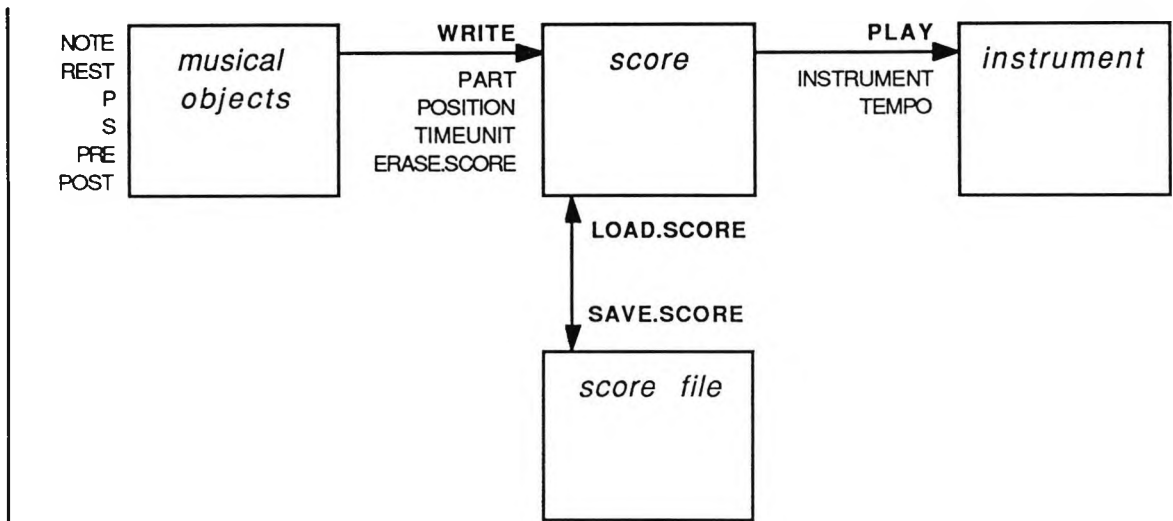


Figure 3. Information flow in the score system.

5 USE OF CHOICE SYSTEMS

Composition can be looked at as the making of choices. Choices can be made by the composer or by the computer. In the last case the composer (you!) will have to give the rules to the computer. One of the possible rules is a random choice from a fixed set of possibilities. We create such a program, a so-called "choice system", by means of the program generator **ALEATORIC**

ALEATORIC

```
ALEATORIC "COLOR [RED GREEN ORANGE]
SHOW COLOR
RED
SHOW COLOR
ORANGE
REPEAT 4 [SHOW COLOR]
GREEN
GREEN
ORANGE
RED
```

Note that the program **COLOR** is just a normal Logo program, except that it is created by a program (by **ALEATORIC**). **COLOR** can be used as any Logo program, like in the example where its result is printed repeatedly. (If you do not know about **REPEAT** you should have a quick look in the Logo manual here.) Note also that **COLOR** does not know anything about colors, it just gives you a word as result.

The name of the program (e.g. **COLOR**) can be chosen freely, but mind not to use **LOCO** primitive names as listed in chapter 10, or any logo primitives (see Microsoft Reference Manual). Also the use of names starting with a dot is discouraged: **LOCO** uses them internally.

In the window called "LOCO Work" an overview of all currently available choice-systems is displayed.

Now we are going to use **ALEATORIC** to construct a musical example, creating a melody with pitches chosen randomly from a fixed set of possibilities.

```
ERASE.SCORE
ALEATORIC "PITCH [60 65 67 69 70]
REPEAT 12 [WRITE [NOTE 1 PITCH 1]]
PLAY
```

Until now, we have been supplying a number as the second argument of **NOTE**. Now the program **PITCH** calculates such a pitch number for us. Those of you who know about scales, and like to experiment with them, can immediately start creating random melodies in major, minor, or other scales, by selecting a suitable list of possibilities for **PITCH** (For MIDI pitch/key numbers see chapter 6.7). Note that when you **PLAY** the score a second time you will hear the same notes, while when you repeat the previous example (writing a new sequence of notes with a random pitch) you will hear a different score.

We also can make random rhythms using **ALEATORIC**:

```
ERASE.SCORE
ALEATORIC "DURATION [1 2 4]
REPEAT 12 [WRITE [NOTE DURATION 60 1]]
PLAY
REPEAT 12 [WRITE [NOTE DURATION PITCH 1]]
```

PLAY

In the last example we combined the two mechanisms of choosing random pitches as well as durations. **LOCO** supports lots of ways of combining very simple mechanisms into complex ones. We encourage the free experimenting with these combinations. They are so manifold that it is impossible to describe them all, just like describing all possible Logo programs is an infinite task.

ERASE.WORK
ERASE.CHOICE

When you want to start all over with a new composition you can use **ERASE.WORK**. It will erase all choice systems. **ERASE.CHOICE** can be used for erasing just one choice system.

```
ERASE.WORK
SHOW COLOR
I don't know how to color
```

ORDERED

If you have a predefined order in mind, in which the choices have to appear, you can use the generator **ORDERED** instead of **ALEATORIC**. **ORDERED** would make the right traffic light, compared to the **COLOR** example we did with **ALEATORIC**, but we will give a musical example here:

```
ERASE.SCORE
ORDERED "PITCH [60 61 64]
REPEAT 9 [WRITE [NOTE 1 PITCH 1]]
PLAY
ORDERED "DURATION [1 2]
REPEAT 12 [WRITE [NOTE DURATION PITCH 1]]
PLAY
```

In the second part of the example we put a duration order of two elements against a melody line of three elements. Can you figure out what goes on?
We can use loudness to accentuate one of the patterns:

```
ORDERED "LOUDNESS [1 0.7]
REPEAT 12 [WRITE [NOTE DURATION PITCH LOUDNESS]]
PLAY
ORDERED "LOUDNESS [1 0.7 0.7]
REPEAT 12 [WRITE [NOTE DURATION PITCH LOUDNESS]]
PLAY
```

RESET.WORK
RESET.CHOICE

Select an instrument on your synthesiser that is sensitive for loudness (for which 0.7 and 1 are distinguishable).
Sometimes you need the possibility to reset a choice system in its initial state, as if you never used it. You can use **RESET.CHOICE** here (or **RESET.WORK** that resets all choice systems).
(In the **LOCO** Work window you see how **LOUDNESS** was generated:
ORDERED "LOUDNESS [1 0.7 0.7]).

```
SHOW LOUDNESS
1
SHOW LOUDNESS
0.7
RESET.CHOICE "LOUDNESS
SHOW LOUDNESS
1
```

A third principle of choice used by many composers is serial choice. A serial choice of a given set of possibilities will first use all of them before one can be

chosen again. Think of a serial choice as a bucket full of possibilities (written on little pieces of paper). Each time a new value is needed, one piece of paper is taken out of the bucket at random without putting it back. Only when the bucket is empty all of the possibilities (pieces of paper) are put back in the bucket and the process is continued.

SERIAL

```
ERASE.SCORE
SERIAL "PITCH [60 63 65]
REPEAT 12 [WRITE [NOTE 1 PITCH 1]]
PLAY
ERASE.SCORE
SERIAL "DURATION [1 2 3]
REPEAT 24 [WRITE [NOTE DURATION PITCH 1]]
PLAY
POSITION 0
REPEAT 24 [WRITE [NOTE DURATION PITCH - 12 1]]
PLAY
```

When we use a serial choice for making rhythms, as we did in the second part of the example above, a kind of feeling for measure is created. This happens because after each $1 + 2 + 3 = 6$ time-units we can be sure a note will start. This even becomes more apparent when a second layer of the same kind of notes is added. This second layer is transposed down by an octave (12 semitones) by using just plain Logo arithmetic. Everywhere in **LOCO** the full power of Logo is available for these kinds of calculations.

Our next example will give a demonstration of the orthogonal nature of **LOCO**: everything can be coupled to everything. We will use a choice system to generate a list of possibilities that another choice system can choose from.

```
ALEATORIC "DIVISION [[3 3][4 2][5 1]]
SHOW DIVISION
[4 2]
SHOW DIVISION
[3 3]
SERIAL "DURATION "DIVISION
SHOW DURATION
2
SHOW DURATION
4
SHOW DURATION
5
RESET.CHOICE "DURATION
ERASE.SCORE
REPEAT 12 [WRITE [NOTE DURATION 50 1]]
PLAY
POSITION 0
REPEAT 12 [WRITE [NOTE DURATION 61 1]]
PLAY
```

In this example serial choice system **DURATION** only consults **DIVISION** when the previous list of possibilities is exhausted. Note that when you use these kind of stacked choices the name of the choice system must be quoted when it is used as an input to another choice system. Otherwise, Logo considers it just a program, calculates its result and gives that as a constant value to the second program generator.

The next example shows another way of stacking choice systems. One choice system can choose the name of another one that has to be evaluated.

EVALUATED

```
ERASE.SCORE
```



```

ERASE.WORK
ALEATORIC "HIGH [60 61 62]
ALEATORIC "LOW [50 51 52]
ORDERED "REGISTER [HIGH HIGH LOW]
EVALUATED "PITCH "REGISTER
SHOW REGISTER
HIGH
SHOW PITCH
61
REPEAT 24 [WRITE [NOTE 1 PITCH 1]]
PLAY

```

REGISTER will return the name of a choice system. **PITCH** will use that name and run it once (in this example it produces a pitch number).

CONSTANT

To make your programs more readable you can use **CONSTANT** to give names to your musical structures, as in the following two examples:

```

ERASE.SCORE
ALEATORIC "PITCH [60 61 62]
CONSTANT "LOUD.NOTE [NOTE 1 PITCH 1]
CONSTANT "SOFT.NOTE [NOTE 1 PITCH 0.6]
CONSTANT "CHORD [P [[NOTE 4 60 1][NOTE 4 63 1][NOTE 4 64 1]]]
REPEAT 12 [WRITE [S [CHORD LOUD.NOTE SOFT.NOTE]]]
PLAY

```

```

ERASE.SCORE
ERASE.WORK
CONSTANT "MINOR [60 62 63 65 67 68 70]
CONSTANT "MAJOR [60 62 64 65 67 69 71]
ALEATORIC "PITCH "MINOR
REPEAT 24 [WRITE [NOTE 1 PITCH 1]]
PLAY
ALEATORIC "PITCH "MAJOR
REPEAT 24 [WRITE [NOTE 1 PITCH 1]]
PLAY

```

TRANSLATED

If you want to use an element by element translation of names you can use **TRANSLATED** that has a list of pairs (a kind of dictionary) as its third input.

```

ERASE.SCORE
ERASE.WORK
ALEATORIC "TONE [C D E F G A B]
TRANSLATED "PITCH "TONE [[C 60][D 62][E 64][F 65][G 67][A 69][B 71]]
SHOW TONE
E
SHOW PITCH
69
REPEAT 12 [WRITE [NOTE 1 PITCH 1]]
PLAY

```

WEIGHTED

When you need choices with different probabilities, you can use **WEIGHTED**. It works like **ALEATORIC**, but assigns to each possibility a different probability. The next example shows the use of weighted durations. Half of the choices will be of duration 1, the rest will be of duration 2 or, with a slight chance, of duration 8.

```

ERASE.SCORE
WEIGHTED "DURATION [[1 0.5][2 0.4][8 0.1]]
REPEAT 12 [WRITE [NOTE DURATION 55 1]]
PLAY

```

SCALED

Sometimes you want to choose from a (very) large number of possibilities. It then can be much easier to give only the borders in between which can be chosen. With **SCALED** you need a minimum, a maximum, and a resolution defining the size of the scaled steps in between (as if we used **ALEATORIC** with a list of equal spaced numbers between a minimum and a maximum). The larger the resolution, the less possibilities there are within the given range.

```
ERASE.SCORE
SCALED "WHOLE.TONE.PITCH 60 72 2
SHOW WHOLE.TONE.PITCH
72
SHOW WHOLE.TONE.PITCH
62
SHOW WHOLE.TONE.PITCH
66
REPEAT 12 [WRITE [NOTE 1 WHOLE.TONE.PITCH 1]]
PLAY
```

A whole tone scale has equal steps between every preceding note. You maybe want to compare the sound and making of this scale with the ones you made with **ALEATORIC**.

When you want to use negative values, mind that Logo is weak in handling negative numbers. Be sure to put parentheses around them, so they will be interpreted in the right way (e.g. (-2)).

The next is an example using **PRE** for precise timing control of a grace note attached to a regular beat, while the duration fluctuates.

```
ERASE.SCORE
SCALED "DURATION 0.1 0.5 0.01
CONSTANT "GRACE.NOTE [PRE [NOTE DURATION 50 1]]
CONSTANT "BEAT [NOTE 2 52 1]
WRITE [REST 1]
REPEAT 12 [WRITE [S [GRACE.NOTE BEAT]]]
PLAY
```

The first **REST** is to prohibit the writing of a **GRACE.NOTE** before the beginning of the score (see explanation of **PRE**).

ITERATIVE

Now we arrive at what we can call higher order choice systems. They are only useful when build on top of other choice systems. A very useful one is **ITERATIVE** which can repeat results a number of times. An example:

```
ERASE.SCORE
ORDERED "PATTERN [64 62 64 67]
REPEAT 16 [WRITE [NOTE 2 PATTERN 1]]
PLAY
ITERATIVE "DOUBLE.PATTERN "PATTERN 2
SHOW DOUBLE.PATTERN
64
SHOW DOUBLE.PATTERN
64
SHOW DOUBLE.PATTERN
62
SHOW DOUBLE.PATTERN
62
RESET.WORK
POSITION 0
REPEAT 32 [WRITE [NOTE 1 DOUBLE.PATTERN + 12 1]]
```

PLAY

Well, in this example a lot of things happened. Let's look at them step by step. The first thing you played was a simple repeated pitch pattern or melody of four notes. The second time the same pattern is doubled, as well as in tempo as in notes. The double tempo is defined by a smaller duration value, but, what is much more interesting, the doubled notes were derived from **PATTERN** by taking its result and repeating every new value two times. Because we checked out **DOUBLE.PATTERN** on the monitor half way, we used **RESET.WORK** to reset **DOUBLE.PATTERN** to its initial state. Otherwise both patterns wouldn't be that synchronous (**DOUBLE.PATTERN**, would start half-way).

PATTERN and **DOUBLE.PATTERN** could also be called Balungan and Pekingan, respectively, if it was a first step in making a simulation of a Javanese gamelan styled composition.

Construct for yourself an iterative choice system that lets another choice system take care of the repetition input (the third input of **ITERATIVE**).

CUMULATIVE

Another higher order program generator is **CUMULATIVE**. Next to the name of the program to generate it has an increment and a start value.

```
ERASE.SCORE
CUMULATIVE "CHROMATIC.PITCH 1 48
SHOW CHROMATIC.PITCH
48
SHOW CHROMATIC.PITCH
49
SHOW CHROMATIC.PITCH
50
REPEAT 12 [WRITE [NOTE 1 CHROMATIC.PITCH 1]]
PLAY
```

Although this is not a very shocking example, **CUMULATIVE** is a strong principle. It, as its name suggests, cumulates all inputs, beginning with its starting value (48 in the example above). The next values are determined by the increment value. That can be a number (as in the previous example), but also, and more interesting, a choice system, as is **THIRD** in the next example:

```
ERASE.SCORE
ALEATORIC "THIRD [3 4 -3 -4]
CUMULATIVE "BROWN.THIRD.PITCH "THIRD 60
SHOW BROWN.THIRD.PITCH
60
SHOW BROWN.THIRD.PITCH
57
REPEAT 24 [WRITE [NOTE 1 BROWN.THIRD.PITCH 1]]
PLAY
```

Brown melodies, as made here with **CUMULATIVE**, have a more flowing nature compared to aleatoric or white melodies.

TRANSITIVE

The next program generator to introduce is **TRANSITIVE**. With this you can link choice systems in a network, each one choosing a next one to use. Like all generators it has the name of the program to be generated as its first argument. The second argument is a starting state.

```
ERASE.WORK
ERASE.SCORE
WEIGHTED "I [[II 0.2][IV 0.6][V 0.2]]
WEIGHTED "II [[I 0.7][IV 0.3]]
CONSTANT "IV "V
```

```

CONSTANT "V "I
TRANSITIVE "PROGRESSION "I
SHOW PROGRESSION
I
SHOW PROGRESSION
IV
SHOW PROGRESSION
V
SHOW PROGRESSION
I
TRANSLATED "CHORD "PROGRESSION [[I C][II D][IV F][V G]]
CONSTANT "C [P [[NOTE 2 48 1] [NOTE 2 52 1] [NOTE 2 55 1]]]
CONSTANT "D [P [[NOTE 1 50 1] [NOTE 1 53 1] [NOTE 1 57 1]]]
CONSTANT "F [P [[NOTE 2 48 1] [NOTE 2 53 1] [NOTE 2 57 1]]]
CONSTANT "G [P [[NOTE 3 50 1] [NOTE 3 53 1] [NOTE 2 55 1][NOTE 3 59 1]]]
REPEAT 12 [WRITE CHORD]
PLAY

```

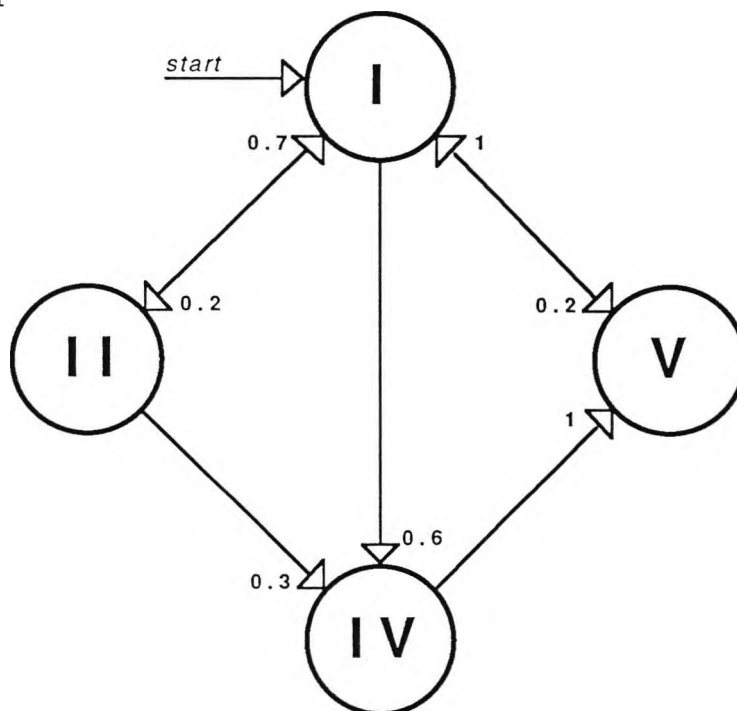


Figure 4. Graphical representation of PROGRESSION.

SAVE.WORK
LOAD.WORK

After all this typing you can save the compositional work by using **SAVE.WORK**. **LOAD.WORK** reads it back in again. (**SAVE.WORK** and **LOAD.WORK** are in fact quite like the Logo primitives **SAVE** and **LOAD**, you can use the File menu items as well (see page 6/7 of the Microsoft Reference Manual). Only **LOAD.WORK** also refreshes the LOCO Work window).

```

SAVE.WORK "PROGRESSION
ERASE.WORK
LOAD.WORK "PROGRESSION

```

SCALE

The last two program generators are used when lists of values are needed instead of isolated values. Sometimes it's useful to have available a scale of values (as used implicitly in **SCALED**). This can for example be used as argument for **SERIAL**. The arguments of **SCALE** are a minimum, a maximum, and a resolution (like **SCALED**).

```

ERASE.SCORE
ERASE.WORK
SCALE "WHOLE.TONE.SCALE 60 72 2
SHOW WHOLE.TONE.SCALE
[60 62 64 66 68 70 72]
ALEATORIC "WHOLE.TONE.PITCH "WHOLE.TONE.SCALE
SHOW WHOLE.TONE.PITCH
70
SHOW WHOLE.TONE.PITCH
64
REPEAT 12 [WRITE [NOTE 1 WHOLE.TONE.PITCH 1]]
PLAY

```

Of course, the arguments for **SCALE** can also be calculated by another choice system.

```

ERASE.SCORE
CUMULATIVE "MAXIMUM 1 60
SCALE "PITCHES 60 "MAXIMUM 1
SERIAL "PITCH "PITCHES
REPEAT 24 [WRITE [NOTE 1 PITCH 1]]
PLAY

```

This example produces a constantly widening serial melody.

COLLECT

If you want to collect a certain number of outcomes from a choice system into a list, it can be passed as an argument to another choice system, using **COLLECT**. In the next example a number of trills are made.

```

ERASE.SCORE
ALEATORIC "PITCH [60 62 64 65 67 69 72]
COLLECT "PITCHSET "PITCH 2
SHOW PITCHSET
[69 62]
SHOW PITCHSET
[60 67]
ITERATIVE "REPEATED.PITCHSET "PITCHSET 4
ORDERED "TRILL "REPEATED.PITCHSET
REPEAT 36 [WRITE [NOTE 0.5 TRILL 1]]
PLAY

```

6 SUMMARY OF COMMANDS

6.1 Writing Scores

ERASE.SCORE

Deletes the entire score

PART *number*

Sets the current part to *number*. *Number* is between 1 and 16 and corresponds to MIDI channels

PART?

Prints the number of the current part

POSITION *number*

Sets current position in the score to a point *number* time-units from the start

POSITION?

Prints the current position in time-units, measured from the start of the score

TIMEUNIT *note-value*

Sets the reference unit of time to *note-value*, a fraction of a whole-note

TIMEUNIT?

Prints the current reference note-value

WRITE *object*

Writes *object* into the score at the current position, in the current part

6.2 Playing Scores

INSTRUMENT *number*

Assigns the instrument to the current part of the score

INSTRUMENT?

Prints the instrument number for the current part

PLAY

Performs the score

TEMPO *number*

Sets the playing tempo to *number* time-units per minute

TEMPO?

Prints the current tempo

6.3 Storing Scores

SAVE.SCORE

Saves the score to disk. Opens a Save dialog box

LOAD.SCORE

Loads a previously saved score. Opens a Load dialog box

6.4 Musical Objects

[NOTE *duration* *pitch* *velocity*]

A note with given *duration* (in time-units), *pitch* (in MIDI numbers), and *intensity* (between 0 and 1)

[REST *duration*]

A rest (silence) lasting *duration* time-units

[S [*object*₁ ... *object*_n]]

A *S*(equence) of objects starting one after another

[P [*object*₁ ... *object*_n]]

A *P*(arallel) structure of objects starting at the same time

[PRE *object*]

A prefix object

[POST *object*]

A postfix object

6.5 Program Generators

ALEATORIC name list-of-possibilities

Name will return a randomly chosen element from the *list of possibilities*

List-of-possibilities is evaluated each time *name* is called

COLLECT name source number

The program *name* will return a list of *number* values from *source*

Source is evaluated once for each *number* of calls of *name*

Number is evaluated each time *name* is called

CONSTANT name value

The program *name* will return value

List-of-possibilities is evaluated each time *name* is called

CUMULATIVE name increment start-value

Each new value that *name* will return is *increment* bigger than the previously returned value. The first value returned is the *start-value*

Increment is evaluated each time *name* is called

Start-value is evaluated at the first time *name* is called

EVALUATED name source

Name will run the choice system given by *source* once

Source is evaluated each time *name* is called

ITERATIVE name source number

Name will repeatedly return a value from *source* a *number* of times

Source is evaluated once for each *number* of calls of *name*

Number is evaluated when all its elements are used

ORDERED name list-of-possibilities

Name will return one of the possibilities in the given order

List-of-possibilities is evaluated when all its elements are used

SCALE name minimum maximum resolution

Name will return an ordered list of random values between *minimum* and *maximum* with the given *resolution*

Minimum, *maximum*, and *resolution* are evaluated each time *name* is called

SCALED name minimum maximum resolution

Name will return a random value between *minimum* and *maximum* with the given *resolution*

Minimum, *maximum*, and *resolution* are evaluated each time *name* is called

SERIAL name list-of-possibilities

Name will return one of the possibilities, excluding it in future choices, until all are chosen. Then it starts all over again with the full list of possibilities

List-of-possibilities is evaluated when all its elements are used

TRANSITIVE name initial-program-name

Each new word that *name* will return is the result of running its previous result as a program. The first result is its *initial-program-name*

Initial-program-name is evaluated at the first time *name* is called

TRANSLATED name source list-of-original-translation-pairs

Name will return a value from *source* translated according to the table (*list-of-original-translation-pairs*)

Source is evaluated each time *name* is called

List-of-original-translation-pairs is evaluated each time *name* is called

WEIGHTED name list-of-possibility-probability-pairs

Name will return one of the possibilities according to the assigned probabilities

List-of-possibility-probability-pairs is evaluated each time *name* is called

6.6 Choice Systems Management

ERASE.CHOICE *choice-system*

Deletes the *choice system*

ERASE.WORK

Deletes all choice systems

LOAD.WORK *file*

Retrieves all choice systems previously saved in *file*. (Identical to the standard Logo primitive **LOAD**, except that it also refreshes the LOCO Work window)

RESET.CHOICE *choice-system*

Resets a *choice system* to its initial state

RESET.WORK

Resets all choice systems to their initial states

SAVE.WORK *file*

Saves all choice systems to a *file*. (Identical to the standard Logo primitive **SAVE**)

6.7 MIDI Pitch Numbers

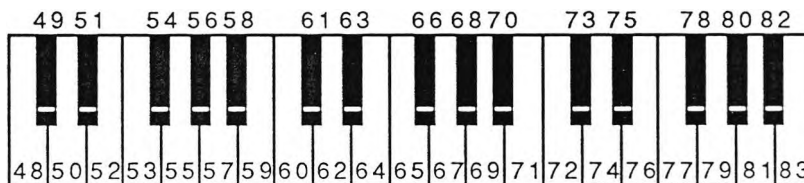


Figure 5. MIDI pitch/key numbers

6.8 MIDI Interface Primitives

MIDIOUT *byte*

Sends a *byte* to the MIDI interface

MIDI.PORT *name-of-port*

Sets the port for the MIDI interface to *name-of-port* (Modem or Printer)

MIDI.PORT?

Prints name of current MIDI interface port (Modem or Printer)

MIDI.SPEED *speed*

Sets the speed of the MIDI interface to *speed* Mhz (0.5, 1 or 2)

MIDI.SPEED?

Prints the current speed of the MIDI interface (0.5, 1 or 2 Mhz)

7 SEPERATE USE OF LOCO COMPONENTS

7.1 Separate Use of the Choice Systems

Choice systems can be used separately e.g. for creating pictures or poems. Load both the files LOCO General Primitives and LOCO Program Generators. Two short examples are listed below.

```
LOAD "LOCO\ General\ Primitives
LOAD "LOCO\ Program\ Generators

SERIAL "ANGLE [-30 -70 -10 20 50 40]
```

Define in the Editor:

```
TO STRANGE.LINE
  REPEAT 6 [RIGHT ANGLE FORWARD 4]
  FORWARD 12
END
```

Now you can type in the text window

```
REPEAT 100 [STRANGE.LINE PU FORWARD 50 PD]
```

Another example with text:

```
ORDERED "SPEECH [I STUT T ER SOME TIMES, AND SOMETIMES NOT]
ALEATORIC "NUMBER [1 1 2]
ITERATIVE "MAX.HEADROOM "SPEECH "NUMBER
REPEAT 12 [TYPE MAX.HEADROOM TYPE "\ ]
I STUT T T ER SOME TIMES, AND AND SOMETIMES NOT NOT
```

7.2 MIDI Output

The MIDI output can be use separately. You can sent an arbitrary byte (number from 0 to 255) through the MIDI interface. In this way you can write programs controlling your synthesiser directly.

The next example shows a procedure that starts your tape recorder ("250" is the MIDI sequence to start a MIDI sequencer):

```
TO START.SEQUENCER
  MIDIOUT 250
END
```

See the manual of your synthesiser for details on the MIDI codes that can be used.

7.3 Score System

The score system can be used separately to make your own music composing program tool. This will give you the score primitives. A quick example:

```
LOAD "LOCO\ General\ Primitives
LOAD "LOCO\ Score\ Primitives
```

Define in the Editor:

```
TO DOWN :FROM :TO
  IF :FROM < :TO [STOP]
  WRITE NOTE 1 :FROM 1
  DOWN :FROM - 1 :TO
END
```

And try out in the text window

```
DOWN 70 50
PLAY
```

8 REFERENCES

- Desain, P. & H. Honing. (1986). LOCO, Composition Microworlds in Logo. In *Proceedings of the 1986 International Computer Music Conference*. edited by P. Berg. San Francisco: Computer Music Association.
- Desain, P. & H. Honing. (1987). LOCO, Composition Microworld, Manual Apple IIe. *Research Report*. Utrecht: Centre for Art, Media and Technology, Utrecht School of the Arts.
- Desain, P. & H. Honing. (1987). LOCO, kompositie mikrowerelden in Logo. In *Leren met Logo 1987*, edited by V. Haars et al. Nijmegen: Leren met Logo.
- Desain, P. & H. Honing. (1988). LOCO: A Composition Microworld in Logo. *Computer Music Journal* 12(3). Cambridge, Mass.: MIT Press.
- Desain, P. & H. Honing. J. Verver (transl.). (1988). LOCO, Kompositie Microwereld, Handleiding Yamaha CX5-MII. *Research Report*. Utrecht: Centre for Art, Media and Technology, Utrecht School of the Arts.
- Desain, P. & J. Henstra. (1988). Audio Illusions in LOCO. *Research Report*. Nijmegen: Katholieke Universiteit.
- Desain, P. & H. Honing. (1989). LOCO, compositie microwerelden in Logo. In *MIDI, Muziek en computer*, by H. Timmermans. Deventer: Kluwer Technische Boeken b.v.
- Desain, P. & H. Honing. (1990). LOCO - eine Kompositionssprache. In *MIDI, Musik und Computer. Theorie und Praxis*, by H. Timmermans. Aachen: Elektor Verlag.

9 INDEX

ALEATORIC 10
COLLECT 17
CONSTANT 13
CUMULATIVE 15
ERASE.CHOICE 11
ERASE.SCORE 5
ERASE.WORK 11
EVALUATED 12
INSTRUMENT
INSTRUMENT? 7
ITERATIVE 14
LOAD.WORK 16
MIDI.PORT 20
MIDI.PORT? 20
MIDI.SPEED 20
MIDI.SPEED? 20
MIDIOUT 21
NOTE 4
ORDERED 11
P 6
PART 7
PART? 7
PLAY 4
POSITION? 5
POST 8
PRE 8
RESET.CHOICE 11
RESET.WORK 11
REST 4
S 6
SAVE.SCORE 5
SCALE 16
SCALED 14
SERIAL 12
TEMPO 5
TEMPO? 5
TIMEUNIT 6
TIMEUNIT? 6
TRANSITIVE 15
TRANSLATED 13
WEIGHTED 13
WRITE 4

10 ACKNOWLEDGEMENTS

We like to thank all our colleagues at the Center for Art, Media and Technology (CKMT of the Utrecht Academy of Arts) for their support and facilities, Sterling Beckwith for general support and suggestions in the naming of the **LOCO** primitives, Irene Lubberink of the **COCO** Foundation, and Eric Brown for advise.

11 USEFUL ADDRESSES

Microsoft Corporation (Logo)
10700 Nothup Way
Box 97200
Bellevue, WA 98009
USA

Applica (MIDI interface)
P.O. Box 1404
NL-6501 BK Nijmegen

Utrecht Academy of Arts
Center for Art, Media and Technology
P.O.Box 1520
NL-3500 BM Utrecht

COCO Foundation
P.O.Box 1037
NL-3500 BA Utrecht

V

CONCLUSION

10

V

CONCLUSION

Towards a general representation of music.....	1
Limitations on making musical knowledge explicit.....	2
Continuous representations underestimated?.....	3
Limitations on the modularization of musical knowledge.....	3
Limitations on the formalisation of musical knowledge	4
Some more concrete issues in the representation of music.....	5
Representation and the problem of consistency.....	5
Representation and ambiguity	6
Representation of articulation and rests	6
A static representation is not enough	6
References.....	8

CONCLUDING REMARKS

In the introduction of this thesis it was stated that it was not the construction of computational equivalents of music cognition that we aimed at, but the development of a language or representational system that enables the description, in clear and precise ways, of the main characteristics of a particular sub-domain of the representation of music, in order to help in the understanding and modelling of cognition. I took an explicit, formal and modular approach to the representation of musical knowledge. This is, as argued before, a restriction for the time being but shown to be one of the better and advisable alternatives to choose. The methodology to obtain these results was to build microworlds dedicated to an isolated problem or a related set of issues. The result being two concrete and successful microworlds: one concentrating on a set of issues related to the representation of time, the other embodying an extensive knowledge representation of structure facilitating the specification of a calculus for expressive timing. They, as such, meet the aim set in the subtitle of this thesis. In conclusion, however, it is good to wonder about the question of what might be lacking if one restricts oneself to such an approach, especially in the research towards a general representation of music.

TOWARDS A GENERAL REPRESENTATION OF MUSIC

The task of constructing a general representation of music is hard to imagine and to plan. Especially since projects of a comparable complexity did not reach high levels of success. We still lack a general theory of representation "a sobering fact since our systems rest on it so fundamentally" (Smith, 1991). General representation languages are still under development, and there are, besides lots of technical difficulties, still theoretical and philosophical problems of enormous proportions. I nevertheless think that it is very important to look for generalizations and abstractions in the representation of music in all its aspects. An alternative position is summarized in the statement "A representation depends on its use" (Roads, 1984; Pope, 1988; Huron, 1990b), a viewpoint described by Christopher Longuet-Higgins (1990) in the following quote:

"My only comment is to remark that the quality of a representation depends on how well it fulfils the purposes for which it is intended, and to underline the need to specify exactly what these purposes are, and how the representation is to be used in achieving them. A blindingly obvious, but by no means trivial, example is the remarkable efficiency of stave notation for the purpose of sight-reading - a form of representation from which we still have a great deal to learn"

Although this is a valid approach to the respective domains of music representation - whether it is in music notation, printing, archiving, the construction of sound and

sequencer files formats etc. -, the aim of constructing a general representation is to bring out the generalizations and abstractions that are not primarily influenced or guided by their use. I prefer this path 'generalization and abstraction' to that of 'dedication and specialisation' (i.e. to design a new and therefore "efficient" representation for every new task or problem) and that forces one to describe what is shared among all these representations.

In the following paragraphs the possible restrictions on the three main aspects of our approach to the representation of music will be discussed (i.e. explicit, modular and formal), hopefully functioning as an appetizer to future work in this field.

Limitations on making musical knowledge explicit

If we look back to the relative complexity of the expression calculus microworld with its explicit structural descriptions, it is hard to imagine the size of a representational system that incorporates all musical knowledge in an explicit manner. It seems that, at a certain stage, implicit knowledge can not be ignored.

Implicit knowledge is frequently used in all kinds of computer systems. Think of a simple library catalogue system. What is often retrieved is implicit knowledge, for example when a user combines facts on the country of publication and the author's year of birth to obtain information on books of Renaissance writers in England - information that is not explicitly represented in the electronic catalogue. But the extraction and representation of implicit knowledge will always be dependent on explicit information (in this example an explicit representation of a book with e.g., a title, the author's name, his/her year of birth, etc.).

At a later stage of building a representational system we might have to consider the notion of skill, also a kind of implicit knowledge (one has to be careful not to treat the entire world as a collection of explicitly representable objects). Skill is acquired by practice and experience and can not be represented explicitly. As an example one might think of expressing knowledge on performing ornaments, how they are played at different tempi and in various musical styles. This knowledge is most readily expressed in a procedural, implicit way, and in this way the explicit structural representation of the calculus provides the hooks to which this implicit behaviour can be attached (a good test for the expressive power of expression calculus and its extendibility).

When a proper set of explicit structural descriptions is given together with powerful ways of providing methods or procedures that express implicit knowledge, and

extendibility is well-supported, one can envision a more or less complete framework for a representational system of music.

Continuous representations underestimated?

However, especially in the case of the continuous aspects of music, we still lack the availability of proper explicit representations that can deal with these continuous aspects in a flexible and a comprehensible way.

Let's take the time functions microworld as an example. The functions that are defined in terms of the generalized time function have, despite their expressive power (e.g. support of function composition and embedded, automatic behaviour), procedural characteristics: after definition they are not accessible (time functions cannot be de-composed) and, as such, are closed to inspection. To have a declarative description in parallel, that allows for this inspection, would combine the advantages of function composition and encapsulated behaviour with the accessibility of a declaratively styled representation.¹

In a larger context, it appears that representations of a continuous nature can improve the flexibility of representational systems considerably. They sometimes yield a level of performance that is not obtained by their discrete counterparts (see e.g. Desain, 1991). For example, in our research on rhythm perception we represented a temporal sequence as continuously variable values that specify event durations, as cells in a connectionist network. This proved a powerful representation for separating the continuous aspects from the discrete aspects in musical time, precisely because of its non-discrete, non-symbolic nature (Desain & Honing, 1989). I would not be surprised if the use of continuous representations prove beneficial in other areas of music perception and cognition as well, because of this flexible and sub-symbolic character from which discrete and symbolic representations may arise. Continuity has been underrated for too long now, both from a technical viewpoint -in many cases considering a discrete representation a harmless simplification-, and from musicological and psychological perspectives which, more or less, overstressed the importance of discrete categories.²

Limitations on the modularization of musical knowledge

The idea that in music "everything has to do with everything", and the impossibility to describe aspects of it in isolation, finds a lot of support in ethno-musicological research. But I think that the perceptual aspects of music as a whole can be profitably understood by describing them in a formalised way, ignoring a larger context (e.g. as in a microworld). One could also argue that music shouldn't be restricted, because if it is, the

restriction would be set apart on purpose by the makers of music (and its active listeners). As David Huron (1990a) has pointed out, it is important how to approach the study of music and how to compare these different approaches (e.g. social, perceptual, historical); a universal representation of music is impossible and the pursue of it should be rejected. Such a universal representation will have "worldly proportions, [...] will change music in unpredictable ways, and there is no neutral point of view from which to begin." And, indeed, a universal representation of music seems impossible. Therefore I prefer to use the term 'general'. The definition of general is important here, since it makes significant restrictions. With 'general' I mean firstly, a representation that describes the measurable and perceptual aspects of music (i.e. a sound signal) and secondly, the cognitive aspects that are directly involved with this perception. The latter is a bit of a problem. The term 'cognitive' refers to models or systems that contain and process knowledge. But are there any limits on the knowledge we need for our 'general' representational system? We have to be able to restrict the required knowledge. I will elaborate on this in the next paragraph.

Limitations on the formalisation of musical knowledge

The viewpoint that "everything is important for a representation of music" in relation to our modular approach in obtaining such a representation, brings us to the "frame problem" (McCarthy & Hayes, 1981): a problem that arises when knowledge has to be encapsulated, separated from the rest of the world knowledge. It is difficult, and most of the time even impossible, to determine what knowledge is affected and what knowledge is unaffected by a certain change or addition of new knowledge to a knowledge base. If we think of a microworld as a small knowledge base, the possibility to extend and combine microworlds can be questioned. A number of philosophers and cognitive psychologists have come up with pro and contra arguments related to this problem (pro: Pylyshyn, Fodor; contra: Dreyfus, Searle). Jerry Fodor takes an important stand in this. He doubts the possibility of formalising cognitive processes. They are part of one central system that is global, non-modular, and therefore cannot - with our current theoretical tools and methods - be comprehended, and can therefore not be formalised. He considers this lack of understanding as the basis of a failure in formalising cognitive processes: "cognitive science has not even started". He thinks the cognitive sciences can be and are successful in formalising the modular parts of the mind: the input systems that are "cognitive impenetrable" (like the five senses and language). These are a successful domain for AI and psychological research (Fodor, 1983).

The problem now becomes whether music can be considered as being part of this central system, or whether it is a module on its own? It clearly is part of the former if one takes

into account all the social and cultural aspects of music; music can be a cognitive faculty among a lot of other things. Restricting a representation of music to, first, all the information measurable in the sound signal itself and secondly, by the cognitive processes that directly interact with it, seems limited enough to gain some level of success (following Fodor's argument). Within this definition I think it is possible to work towards generalizations that can form a basis of cognitive models of important aspects of music. A positive consequence of such a 'decontextualized' representation is its effectiveness in a carefully restricted domain where almost all the knowledge is special to that domain (i.e. little or no common-sense knowledge is required). The question, whether music cognition can be described as such a restricted and isolated domain, is still open.

SOME MORE CONCRETE ISSUES IN THE REPRESENTATION OF MUSIC

After these ideas of a mostly philosophical nature, I will, finally, return to a list of more concrete problems that should be explored in the near future.

Representation and the problem of consistency

Consistency is a relational property. It describes the organizational principles of a set of rules or statements, or, in other words, it relates a set of statements or rules to the set as a whole. Without these organizational principles the number of possible different configurations meeting the structural requirements would be quite large. Consistency, therefore, is an essential characteristic of a formal system. But what are the demands on consistency of a representational system for music?

In the calculus for expressive timing structural consistency is a given; the structural descriptions are not changed as a result of a transformation applied to the representation. But, when we have to loosen this restriction in a higher-level representation based on such a calculus, structure changing transformations should be possible. For example, a chord could be changed into an by applying a *ritardando* transformation and the rhythmical structure could also change because of such a transformation.³ In (Honing, 1991) it has been proposed to use constraints on time intervals to distinguish between similar structural objects, with the coordinated behaviour imposed by these constraints modelling their specific structural character.⁴ On the basis of these descriptions, a parse mechanism could take care of situations where transformations on a given musical object would change its organisation (e.g. changing a chord into an arpeggio), as a result delivering an updated structural description, i.e. a different set of relations associated with the transformed musical object (a set of

arpeggio constraints replacing the set of chord constraints), changing the behaviour of the structural unit under future transformations. In this way, a representational system could incorporate structure changing behaviour.

Representation and ambiguity

How can ambiguity be described in a representation system? How to coordinate the interaction between ambiguous structures and their associated behaviour? For example, we need a representation formalism that can express the ambiguous structure of a certain musical fragment and the way it influences the expressive timing profile in a performance.

Another problem, often confused with ambiguity (I was no exception), is the notion of overlapping structure, like overlapping phrases, where, for instance, one note is part of both the first and of the second phrase. This type of phrase structure is not ambiguous, ambiguous in the sense that there is *more than one* structural description possible, since it is best described as *one* structural description that comprises the dependencies of more than one overlapping (or cross-branching hierarchical) descriptions. This is also an acknowledged representational problem in phonetics where one phoneme can be part of two syllables at the same time, and in spoken language, where sentences like "I think, John is over there, I think" are not exceptional. It might turn out that the support for "overlapping structure" is a key aspect in the representation of music since it conflicts with the description of music as a formal grammar of rewrite rules finding such a widespread application.⁵

Representation of articulation and rests

How can the difference between articulation and rests be represented? In certain contexts, a note and a silence can be perceived differently. The structure can either be described as a note and a rest, or as a staccato note without a rest. There are several theories that try to explain this, on a discrete basis (Longuet-Higgins, 1984) and on a continuous basis (Desain, in preparation). It seems that a solution should be formalised in terms of the surrounding structure, like the metrical and rhythmical structure and, of course, the absolute tempo.

A static representation is not enough

Especially with regard to the last question, we need the introduction of theoretical notions like expectation and attention, and principles that have to do with the process of

building up and determination of structure, based on a process-oriented description of music perception. The representation issues described in this thesis could ignore (for the time being) these process-oriented descriptions because they were not aimed at the modelling of human cognition. All too often, though, process-like descriptions are preferred over static ones because of this valid psychological argument. However processes like quantization, closure, planning, etc., need some overview, and therefore static descriptions (Desain & Honing, in preparation). A balance between static and dynamic process-oriented descriptions is important here. Processes are made up of a knowledge representation and an algorithm, with the algorithm depending heavily on the representation chosen (cf. implicit knowledge discussion above). Therefore, moving the focus to the process aspects of a representational system will most likely influence the proposed representations (hopefully) in the form of extensions or generalizations. But here, once again, the first task is to make these processes explicit, in a modular and formal way, instead of leaving them in procedural obscurity.

• • •

This list of points concerning the generalization of a representation of music could be continued but it would be accompanied by more and more speculation and less and less justification. So this seems to be the right moment to finish this thesis, a thesis in which I hope to have shown, at least, a successful integration of knowledge representation research with work in the field of the psychology of music.

REFERENCES

- Desain, P. & H. Honing (1989) Quantization of Musical Time: A Connectionist Approach. Computer Music Journal 13(3). Cambridge, Mass.: MIT Press: 56-66.
- Desain, P. & H. Honing (in preparation) Must "real-time" equal "no-idea-of-time" in composition systems?
- Desain, P. (1991) A Connectionist and a Traditional AI Quantizer, Symbolic versus Sub-symbolic Models of Rhythm Perception. In: Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press.
- Desain, P. (in preparation). Metre as a continuous concept.
- Fodor, J. (1983) The Modularity of the Mind: An Essay on Faculty Psychology. Cambridge, Mass.: Bradford Books, MIT Press
- Honing, H. (1991) Issues in the Representation of Time and Structure in Music. In: Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press. [This thesis].
- Huron, D. (1990a) *Personal Communication*. A letter commenting on Honing, 1991, and in response to a letter by the author of this thesis commenting on Huron, 1990b.
- Huron, D. (1990b) Design principles in computer-based music representation. In: Computer Representations and Models in Music, edited by A. Marsden and A. Pople. London: Academic Press.
- Longuet-Higgins, H.C. (1987) Mental Processes. Cambridge, Mass.: MIT Press.
- Longuet-Higgins, H.C. (1990) *Personal Communication*. A letter commenting on Honing, 1991.
- McCarthy, J. M. & P. J. Hayes (1981) Some philosophical problems from the standpoint of artificial intelligence. In: Readings in Artificial Intelligence. Palo Alto: Tioga Publishing: 431-450.
- Pope, S. T. (1988) Music notations and the representation of musical structure and knowledge. Perspectives of New Music 24: 156-189.
- Roads, C. (1984) An overview of music representation. In: Musical Grammars and Computer Analysis, edited by M. Baroni and L. Callegari. Firenze: Olschki: 7-37.
- Smith, B. C. (1991) The owl and the electric encyclopedia. Artificial Intelligence. 47: 251-288.

NOTES

- ¹The use of multiple representations was discussed earlier as a possible solution, a solution, though, that has not yet been investigated in the case of the generalized time function.
- ²This is not to say that discrete elements do not play a central role in e.g. music perception, but to stress the loosely defined hypothesis that a continuous basis might explain this discreteness even better, in a more flexible and complete way.
- ³Though, in the 'real' world, musicians are often very good in the application a "correct" or right amount of *ritardando* in a way such that the rhythmical structure is *not* lost.
- ⁴Unfortunately, although planned, these ideas have not yet been realised. Some of the ideas have been loosely tried out and are planned to be incorporated in a system for computer animation called *Cocoa*.
- ⁵These ideas are the result of several discussions with Peter Desain on this problem.