THESE de DOCTORAT de l'UNIVERSITE PARIS 6

Spécialité :

Acoustique, Traitement de signal et Informatique Appliqués à la Musique

Sujet de la thèse :

DESIGN AND IMPLEMENTATION OF AN INTEGRATED ENVIRONMENT FOR MUSIC COMPOSITION AND SYNTHESIS

(CONCEPTION ET REALISATION D'UN ENVIRONNEMENT INTEGRE POUR LA COMPOSITION ET LA SYNTHESE MUSICALE)

Thèse présentée par M. Peter Hanappe pour obtenir le grade de DOCTEUR de l'UNIVERSITE PARIS 6 devant le jury composé de :

MM.	Emmanuel Saint-James,	Directeur
	Peter Desain,	Rapporteur
	Andrzej Duda,	Rapporteur
	Gérard Assayag,	Examinateur
	Claude Girault,	Examinateur
	François Pachet,	Examinateur
	Christian Queinnec,	Examinateur

Thèse préparée à l'Ircam – Centre George-Pompidou,
Equipe Représentations Musicales,
sous la responsabilité de Gérard Assayag.

Acknowledgments

I thank Peter Desain and Andrzej Duda who accepted to report on this work. I would like to thank Claude Girault, François Pachet, and Christian Queinnec who are willing to be members of the jury.

Special thanks must go to Gérard Assayag who accepted me in his team during more than three years. His ideas and vast musical culture enriched this work; his generosity and support have been exemplary.

I thank my supervisor Emmanuel Saint-James. Although we work in different places he closely followed my work. His comments and suggestions on this work have been invaluable.

I would like to thank the professors of the DEA ATIAM for accepting me to their classes. Thanks to Jean-Claude Risset and Philippe Depalle.

The fantastic team spirit has been a major stimulans for my work. Many thanks and coffees to Carlos Agon, Claudy Malherbe, Camilo Rueda, Olivier Delerue, and Benoit Meudic.

I thank Ircam and the many people that helped me there. Thanks to Hugues Vinet, Marie-Therese Join, Sylvie Benoit, Dominique Doublet, Laurent Ghys, Eric Daubresse, and Serge Lemouton.

I would like to thank Brad Garton (Columbia), Robert Rowe (NYU), and Daniel Oppenheim (IBM) for their hospitality and exchange of ideas during several trips to New-York.

The discussions with Marco Stroppa, Gerhard Eckel, Ramon González-Arroyo have been very enriching. Artistic projects of projects inspired my work. Thanks to Didier, Sylvie, Erik, Alain, Renny. Special thanks to Daniel and Sam for reformulating my thesis.

I would to like to thank the friends for bearing with me the last three years: Laurent "Brain", Bennet, Adrien, Franscisco, Federico, Geoffroy, Christophe, Richard, Tom, and Gilbert.

Finally, I would like to thank my parents. Without their support there would have been no expedition to Paris and no finished thesis.

Abstract

In this text we present an integrated environment for computer music. Our system integrates a high-level Scheme interpreter, a real-time synthesizer, and a rich framework for temporal composition and sound synthesis. The environment is entirely written in the Java programming language and can be used in distributed applications. Three aspects of computer music that are generally treated separately – composition, sound synthesis, and interactivity – are tightly integrated in this environment.

The embedded Scheme interpreter offers an interactive programming environment. We show how the underlying Java platform promotes a transparent use of functional Scheme objects throughout the system. These functional objects, which we called *programs*, are used to describe the complex behaviors of the system. The events, for example, carry with them a program to describe their actions.

The compositional structures organize both discrete elements and continuous control functions in a hierarchical structure. Compositions thus become complex descriptions that control the sound synthesis. The basic element of temporal composition is the *activity*. *Patterns* organize activities in time and maintain the temporal relations between them. Changes to the organization are updated incrementally. This resembles the techniques of constraint propagation found in graphical interfaces. Causal relations can be used to describe the organization of activities of unknown duration.

The basic unit of sound generation is called a *synthesis process*. They are created with a meta-class approach using *synthesis techniques* and *synthesis voices*. Synthesis processes are aware of the time relations defined in the composition. This time information is bundled in an object called *time context*. Synthesis processes can use this information to deform the real time displayed by the synthesizer.

The environment concurrently handles the Scheme interaction, the garbage collection, and the real-time synthesis. We investigate whether hard real-time can be guaranteed in such a dynamic environment. It is difficult to conclude on the question solely on the basis of the discussions found in the literature. However, we introduce a constraint on the synthesis processes that reduces the question to the scheduling problem of concurrent tasks.

Résumé

Nous présentons un environnement intégré pour la composition et la synthèse musicale sur ordinateur. L'architecture que nous proposons combine dans un seul système la synthèse en temps-réel, une gestion de la memoire dynamique avec un ramasse-miettes, le traitement interactif d'événements, et le dialogue avec l'utilisateur à travers un interprète Scheme. L'environnement est entièrement écrit dans le langage de programmation Java et peut être utilisé dans des applications réparties. Trois aspects du domaine de la musique sur ordinateur – composition, synthèse et interactivité – sont intégrés de manière intime dans cet environnement. Les concepts et les structures décrits dans cette thèse peuvent être étendus et appliqués plus généralement aux environnements multimédia.

L'interprète Scheme embarqué offre un environnement interactif de programmation. La plate-forme Java sous-jacente promeut l'utilisation transparente des objets fonctionnels dans le système. Ces objets fonctionnels, que nous appelons *programmes*, sont utilisés pour décrire le comportement complexe du système. Les événements, par exemple, sont composés d'un temps d'évaluation, d'un programme et des arguments du programme. Au temps indiqué le programme est appliqué avec ses arguments. Ainsi, l'interprète permet à l'utilisateur d'exprimer et modifier les actions du système dans un langage de haut-niveau.

Nous avons nommé l'unité de base de la composition une *activité*. Elle est est composée d'une position dans le temps, d'une durée, d'un programme de début et d'un programme de fin. Les deux programmes décrivent respectivement les actions à prendre pour commencer et terminer cette activité. L'organisation d'activités dans le temps est prise en charge par une structure appelée *pattern*. Les patterns regroupent un ensemble d'activités ainsi que les relations temporelles entre elles. Ces relations sont vérifiées après chaque manipulation de temps et une réorganisation des positions et des durées des activités peut en résulter. Cette stratégie se rapproche des techniques de la propagation de contraintes que l'on trouve dans la construction des interfaces graphiques. Les *motifs* permettent d'insérer un pattern à plusieurs reprises dans une composition. L'organisation des activités de durée inconnue au moment de la composition peut être réalisée à l'aide des operateurs d'intervalles basés sur des relations causales.

Les patterns et les motifs permettent la structuration d'une pièce de manière hiérarchique. A chaque niveau de cette hiérarchie nous attachons un axe de

temps local. L'axe de temps d'un niveau est projeté sur l'axe du niveau supérieur moyennant un *modèle de temps*. Ces modèles permettent la déformation du temps local.

L'objet unité pour la génération de son est appelé un *processus de synthèse*. Il est créé en utilisant une stratégie de méta-classes : une *technique de synthèse* crée une nouvelle *voix de synthèse*, la voix de synthèse crée un nouveau processus de synthèse. L'information temporelle codée dans la partition est rendu disposible aux processus de synthèse. Cette information est regroupée dans un objet appelé *contexte de temps*. Il assure, entre autre, la conversion du temps indiqué par le synthétiseur dans le temps local d'un processus de synthèse.

L'interactivité du système est rendue possible grâce à la gestion concurrente de la synthèse en temps-réel, du traitement des événements et de l'entrée de données textuelles dans l'interprète Scheme. Dans cette thèse nous évaluons si un environnement dynamique, tel que nous le présentons, intègrant un ramasse-miettes, peut garantir une exécution en temps-réel "dur." Il est difficile de répondre à cette question seulement à partir des discussions qu'on trouve dans la littérature. En revanche, nous introduisons une contrainte imposée sur les processus de synthèse qui réduit l'interaction entre le ramasse-miettes et la tâche temps-réel au problème du scheduling de tâches concurrentes.

La thèse est divisée en deux parties : la première présente l'état de l'art, la deuxième présente notre travail. La partie concernant l'état de l'art contient les chapitres suivants :

Chapitre 1 discute les techniques de programmation devenues standard, en particulier les notion de temps-réel dur et douce, la gestion de la mémoire dynamique, les ramasse-miettes, et la gestion de multiple tâches.

Chapitre 2 donne une brève introduction au domaine de la musique sur ordinateur. Nous abordons le sujet de la synthèse sonore et la composition assistée par ordinateur.

Chapitre 3 détaille en plus la question de l'organisation temporelle dans la composition. Nous abordons le problème de l'organisation des éléments discrets et l'utilisation des fonctions continues.

La partie concernant le travail effectué contient les chapitres suivants :

Chapitre 4 présente les bases de l'architecture de l'environnement que nous proposons dans ce travail. En particulier, nous argumentons le choix des langages, les concepts de bases tels que les événements, les programmes et les processus de synthèse. Nous donnons une description formelle de l'architecture.

Chapitre 5 détaille la réalisation de l'architecture ainsi que les classes de base pour la synthèse sonore. Nous montrons également comment l'environnement peut être utilisé dans une application répartie.

Chapitre 6 introduit les structures pour l'organisation temporelle d'une composition. Nous expliquons l'utilisation des patterns, motifs, modèles de temps et contextes de temps.

Chapitre 7 analyse le comportement temps réel du système. En particulier, nous estimons l'influence du ramasse-miettes sur le comportement de la tâche temps réel.

Contents

List of Figures

Introduction

In this text we present an integrated environment for music composition and performance. Our system integrates a high-level garbage collected Scheme interpreter, a rich time model, a real-time synthesizer metaphor, and distributed input and output. We discuss a theoretical model as well as the implementation details.

The bulk of musical applications still approach music composition and sound synthesis as two distinct domains within the realm of computer music. This separation may be traditional, as music has always a two step process of composition and performance. However, when musicologists mention two distinct tendencies in the music of second half of the twentieth century – the first preoccupied with new techniques of music writing, the second preoccupied with the organization of sound itself – it might be due to the lack of tools to bridge the two fields. Many composers have argued to break this pattern and sought to extend musical thought to all levels of a music piece, from form to sound. Already in 1917 Edgar Varèse wrote:

> I dream of instruments obedient to thought - and which, supported by a flowering of undreamed-of timbres, will lend themselves to any combination I choose to impose and will submit to the exigencies of my inner rhythm. ([Var72], cited in [MMR74]).

Essential in the design of a music system is the choice of a computer language and the specification of a time model. The choice of a language is important for several reasons. Firstly, the language must allow an efficient implementation of signal processing techniques. Since we want to realize a real-time synthesizer, sound synthesis must be performed within predictable delays. Secondly, the environment must be easily extendible. We provide a basic set of synthesis modules and control functions. Expert users will likely want to extend the environment with new modules. Thirdly, the user must be able to "program" the environment. The definition of synthesis networks, musical structures, and the system's behavior in an interactive set-up escapes any trivial description. The only way to communicate such complex information is thru the use of a language.

We have chosen to develop the environment in the Java programming language. In addition, we embed a Scheme interpreter into the system for the user interaction. This high-level programming environment allows the user to extend the set of primitives of the environment in ways that were not foreseen by the programmer. Since the Scheme interpreter is implemented on top of the Java platform, one single object system and one single memory strategy is used in the environment. This promotes a transparent use of functional objects throughout the system. In particular, functional objects are used extensively to describe complex behaviors and relations. Events, for example, carry high-level functional descriptions of their side effects and provoke the application of composition algorithms.

The choice of Java as programming language raises a number of technical questions. In particular, it places a garbage collector concurrent to a real-time task in a multi-threaded environment. In this thesis we estimate if hard real-time in such an environment is possible at all. We impose the constraint that the synthesizer thread can not allocate storage space. This is not a big constraint since synthesis techniques rarely allocate new objects. Under the conditions specified in the text, the problem of the garbage collector can be reduced to that of concurrent tasks. The latter is a well-documented problem in a real-time system design.

Time is undoubtedly the most important dimension in music. A rich time model is therefore essential in composition environments. On the one hand, most composition environments only treat discrete time. However, most parameters of the sound synthesis vary continuously in time. We thus need continuous time functions for the control of the sound synthesis. On the other hand, most synthesis programs offer few tools to organize music pieces. Our environment integrates a continuous time model into a hierarchical description of the music structure. This layered, continuous time model allows the expression of non-trivial time dependencies.

In our attempt to integrate composition and synthesis we realized that most synthesis systems have a very narrow notion of time. If we want to allow real-time interaction with compositional structures and assure that the system responds consistently, we must make the system aware of the complex time relationships in the composition. In addition, statical structures cannot respond to user input. To extend the composition process to more dynamic pieces we have to include behavior and causal relations into the final score.

With the proposed framework we give the composer a set of tools for the control of sound synthesis. Since, both discrete elements and continuous control functions are organized in a single hierarchical structure, compositions become complex organizations that control the sound synthesis. Control functions can be generated by the same compositional process that generates the musical form. As such, the defition of the timbre becomes an integral part of the composition.

The proposed system offers an increased flexibility to integrate composition, synthesis, and interactivity. We pretend that our system overcomes the still existing barrier between "music writing" and "sound sculpting." Our environment allows the composer to travel seamlessly between different layers of abstraction, from the "sub-symbolic," concrete level of sound synthesis to the symbolic, abstract level of musical form. Decisions on any level of the composition may influence the organization on any other level. During runtime, high-level compositional processes can interact with low-level synthesis processes. Vice versa, low-level synthesis processes may guide compositional algorithms. In addition, user events may cause a complete reorganization of the composition during the performance. We think that a rigorous implementation of these ideas can lead to new inspiration in musical writing.

Although this work started from a reflection on music systems, some of the concepts discussed in this text can be applied more generally to all time-based media. In particular, animation systems could use a layered representation of time to their advantage. Interactive graphics systems could use an embedded Scheme interpreter and rich composition structures to offer more intelligent interactions than chaining together pre-recorded sequences. Where we can in this text, we will abstract the musical origin of our research and consider time-based media in general.

Our work is more practical than theoretical; application design has been our major concern. We consider our project as an integration of existing ideas. Our research project touches many fields including garbage collection techniques, real-time scheduling, distributed systems, dynamic languages, embedded interpreters, sound synthesis, interval algebra, temporal constraints networks, and algorithmic composition. Notions in each of this fields are required to realize our project: an integrated environment for music composition and synthesis.

Overview of the Thesis

This thesis is organized in two parts. In the first part we give an overview of the state of the art. We start with a discussion of programming techniques and operating system designs relevant to the remainder of the text (Chapter 1). In particular, this overview will help us in the analysis of the real-time performance of the proposed environment. We then give an introduction to the field of computer music (Chapter 2). We dedicate a separate chapter to the representation of time and structure in composition environments (Chapter 3).

Part two presents our work. First, we discuss the fundaments of the architecture: the choice of Java as development language, the use of and embedded Scheme interpreter, the basic concepts, and the formal model that serves as the basis of our system architecture (Chapter 4). Implementation issues and basic classes are discussed in the following chapter (Chapter 5). How the proposed environment represents time and structure can be found in chapter 6. Finally, we examine the real-time performance of the environment (Chapter 7).

State of the art

Chapter 1

Standard designs for real-time programs, dynamic memory management, and multi-tasking

Although computer science is constantly evolving and new programming techniques are proposed frequently, several techniques have become standard. Since the discussion of our environment will depend on these techniques, will present them early on in our text. The reader familiar with these techniques can jump this chapter without risking to miss something of our presentation.

Computer systems manage two major scarce resources: the storage space and the CPU time. Other scarce resources include network and disk bandwidth. If a program has to output its results within well-defined time delays, it is called real-time. Real-time programs, in collaboration with the operating system, have to balance their CPU usage carefully. Section 1.1 discusses some issues in the design of real-time systems. Programs whose storage needs vary during the execution must take special measures to prevent the exhaustion of memory space. Section 1.2 is dedicated to techniques of dynamic memory management. The number of tasks that a computer system can handle truly simultaneously is equal to the number of CPU's in the system. Section 1.3 discusses how the system can simulate concurrent activities with the use of processes and threads.

1.1 Hard versus soft real-time applications

In this section the reader will find a discussion of the difference between "hard" real-time and "soft" real-time. Real-time engineers often distinguish between hard and soft real-time. In hard real-time, a correctly computed result delivered after its deadline is as incorrect as an incorrectly computed result. The system is called hard if "the system shall not miss a deadline." In soft real-time systems, it is desirable to deliver all computed results prior to their deadlines, but a result delivered late is better than no result at all. The system is soft if "the system should not miss a deadline." The term "real-time" is often misused to indicate a fast system.

Existing real-time systems often comprise hard and soft real-time components. When developing soft real-time systems, we have the option of entirely ignoring traditional methodologies for the design and analysis of real-time systems. The code is optimized and the system is expected to run fast enough to meet the requirements. An occasional missed deadline does not lead to catastrophic results. However, it is desirable to use established real-time methodologies to characterize the performance limitations under various work loads, input conditions, applications, and architectures.

An overview of real-time design methodologies is out of the scope of this text. The general approach is to submit the system to a systematic analysis of its behavior. In particular, the worst case execution times of all instructions and function calls are determined statically thru a code analysis. The sum of these worst case times results in the worst case execution time of the system. The design of hard real-time systems only considers worst execution times, not average times, since no delay can be tolerated.

1.2 Dynamic memory management

The memory management of an application is said to be dynamic if its storage needs vary during the execution. The application then allocates space in the heap to satisfy new storage requests. Since the size of storage space is bounded, the application must free unused objects to assure that future requests can be satisfied. Storage space can be freed either explicitly or implicitly. When explicit memory reclamation is used, it is up to the application developer to free previously allocated objects. When the space is reclaimed implicitly a *garbage collector* searches and frees the objects which are no longer used by the application. Applications written in the C/C++ language typically use explicit memory reclamation. Lisp-style languages and Java rely on a garbage collector to reclaim storage space.

With implicit memory management, the developer need not worry about the bookkeeping of previously allocated memory. When the application runs out of storage space a garbage collector reclaims the allocated memory that is no longer used (hence garbage) by the application. This makes programming simpler and more straightforward. Objects that are still used by the application are called "live." The collection of garbage normally proceeds in two steps: 1) distinguishing the live objects from the garbage (garbage detection or tracing), 2) reclaiming the garbage objects so that their occupied space can be reused (garbage reclamation). The garbage collector traces the live objects starting from the active variables. These includes the statically allocated and global objects, the local variables on the stack, and the variables in the registers. These objects are called the *root set*. Any object that is reachable from the root set by dereferencing is also reachable by the application. It is therefore considered alive. The objects that are not reachable by traversal of the application's data structures are considered garbage.

Both explicit and implicit memory reclamation have their advantages and inconveniences. Explicit reclamation can lead to hard to find errors. Forgetting to free memory leads to an accumulation of garbage (memory leaks) until the application runs out of memory. Reclaiming memory too early can lead to strange behavior. The freed memory can be used to allocate a new object; two objects occupy the same physical space and overwrite each other data causing unpredictable mutations. These errors are hard to find. Moreover, it's possible that the application runs correctly in most situations and that the errors only show up in particular situations. Garbage collection, however, is able to reclaim most garbage and therefore eliminates the risk of running out of memory due to memory leaks.

Garbage collection is often considered expensive, both in CPU time and in storage space. For every object the collector may allocate an additional header to store information about the object. Furthermore, traditional collectors interrupt the application when it runs out of memory to perform a full garbage detection and reclamation. This interruption can take a fair amount of time and result in a poor responsiveness of the application. More advanced garbage collectors using some of the techniques discussed later are sometimes cheaper, are less disruptive, and are usually competitive with explicit deallocation [Zor93, NG95].

In terms of software design, the convincing arguments in favor of implicit memory management is that garbage collection is necessary for fully modular programming. The "liveness" of an object is a global property which depends on the liveness and state of other objects. With implicit storage reclamation, the developer of a routine can reason locally: "if I don't need this object, I don't need reference it anymore." With explicit storage reclamation the developer has to ask the question whether the object in question is still needed by other routines or whether the object can be freed. This introduces non-local bookkeeping into routines that otherwise would be locally understandable and flexibly composable. Determining the liveness "by hand" in the case of explicit storage reclamation becomes more difficult as the application grows. More important, it creates interdependencies between the different modules (data structures, libraries). The user of a library must know by who and when objects are allocated and freed. It is, therefore, not uncommon that the developer falls back on variants of reference counting or other garbage collection techniques to help him with the bookkeeping of unused objects. The integration of an optimized garbage collector for the design of large, dynamic, and modular applications becomes the better choice both in terms of development efforts and runtime efficiency [App87, Zor93].

Before we discuss existing garbage collection techniques, we briefly recall the notions of fragmentation and locality of reference. When objects are allocated and freed dynamically, free space gets interleaved with live objects and the heap space gets fragmented. Without special care the maximum size of contiguous free space may become too small to satisfy allocation requests, even though the total size of free space is large enough. Locality of reference means that if a memory page is referenced, chances are that the next reference will be in the same page. A good locality of reference is important to obtain good performance when virtual memory is used. Part of the virtual memory is kept on the hard disk. Referencing a page that is not loaded but stored on the hard disk results in a page fault. The page will be loaded into memory. When objects of the same age end up in different pages the assumptions about locality of reference do not hold. This will result in many page faults and poor performance. Fragmentation and locality of reference are two factors that influence the efficiency of a collection technique.

In the rest of the section we will give an overview of garbage collection techniques. This brief and incomplete discussion will help us understand the behavior of our system in chapter 4. Much of this section is based on Paul Wilson's overview of uniprocessor garbage collection techniques [Wil94].

1.2.1 Basic garbage collection techniques

We will discuss four garbage collection techniques: reference counting, mark-and-sweep garbage collectors, copy collectors, and non-copying implicit collectors.

Reference counting is not always considered to be a true garbage collection technique. We include it for completeness. In this technique, every allocated object has an associated count. Every time a variable is set to point to an object, the object's count is incremented. When the reference is destroyed, the count is decremented. When the count goes to zero the object can be freed since no variables refer to it. The object is scanned for pointer variables and all the objects to which it refers have their reference count decremented. One advantage of reference counting is that it is "incremental:" the garbage collection is closely interleaved with the program execution and advances step-wise. Updating the reference count takes a few operations that can be bounded in time. The reclamation of objects can be deferred by linking them in a free-list and freeing them few at a time. There are two major problems with reference counting. The first is that reference counting fails to reclaim cyclic structures. The reference counts of the objects that form a cyclic structure are never set to zero. This can lead to an accumulation of garbage. The second problem with reference counting is that its cost is generally proportional to the amount of work done by the running program, with a fairly high large constant of proportionality.

A mark-and-sweep collector proceeds in two phases as in the general case stated above. The garbage detection is called the mark phase, the garbage reclamation the sweep phase. During the mark phase the live objects are traced by traversing the application's data structures starting from the root set. Live objects are marked, often by toggling a bit in the header of the object. Once the live

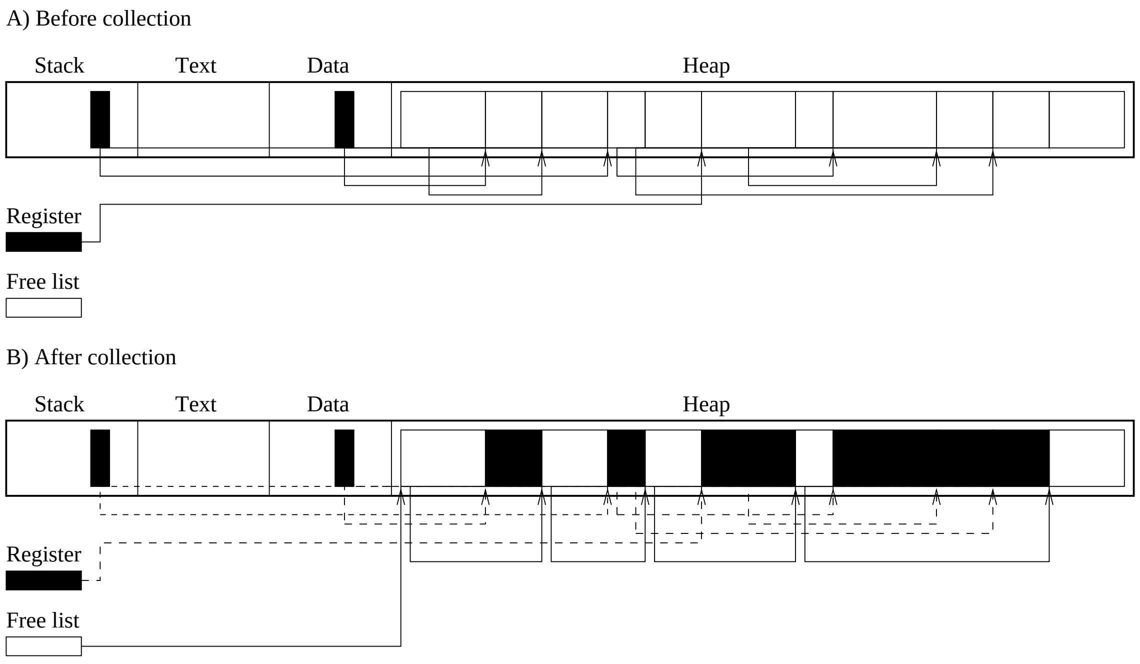

Figure 1.1: *The mark-and-sweep collector: the collector traces the live objects starting from the root set. The root set includes the variables on the stack, in the data segment (global and static variables), and in the registers. The traversal marks the live objects (black in the figure.) The objects that are not marked (white) are linked into a free list.*

objects have been made distinguishable from the garbage objects the memory is swept: all the objects that are not marked (the garbage objects) are linked into a free list (Fig. 1.1). There are three major problems with mark-and-sweep collectors: fragmentation, efficiency, and locality. Since the heap is never compacted objects stay in their place. New objects will be allocated in the free space between old objects. This has negative implications for fragmentation and locality of reference. Mark-and-sweep collectors are therefore often considered unsuitable for virtual memory applications. The mark-compact collector remedies the fragmentation and locality problem of the mark-and-sweep collector. After the marking phase it compacts the heap. Live objects are slid to one side of the heap adjacent one to another. After compaction the free space forms one contiguous space at the other side of the heap. The second problem of mark-and-sweep collectors is that the amount of work is proportional to the size of the heap. To find the garbage the complete heap has to be scanned.

The third type of collectors we discuss is the copy collector. The simple model of a copy collector is the stop-and-copy collector. This collector uses two semi-spaces, the *to-space* and the *from-space*. New objects are allocated in the from-space. When no more space is available all live objects are copied to the to-space. This can be done in one traversal. The from-space is now known to contain only garbage, and the roles of the two spaces are inverted. The term *scavenging* is sometimes used to denote this traversal since only the worthwhile objects amid the garbage are saved. If an object can be reached following two different paths it must be copied only once. To this effect extra space is reserved in the header of an object to store a forwarding pointer. When an object is copied the forwarding pointer is set to the address of the new object. During the scavenging the forwarding pointer is checked first. If the object has already been copied only the pointer that led to the object needs to be updated (Fig. 1.2). The advantages of the copying collector are that allocation can be done very fast. It requires to increment the pointer that points to the available space in the from-space. The copying also results in a defragmentation. Fast algorithms exist to traverse the heap (Cheney's algorithm). The amount of work that needs to be done is proportional to the amount of live objects. Copy collectors do not need to traverse the entire heap, as is the case for mark-and-sweep collectors, which has positive implications for virtual memory applications. A simple way to decrease

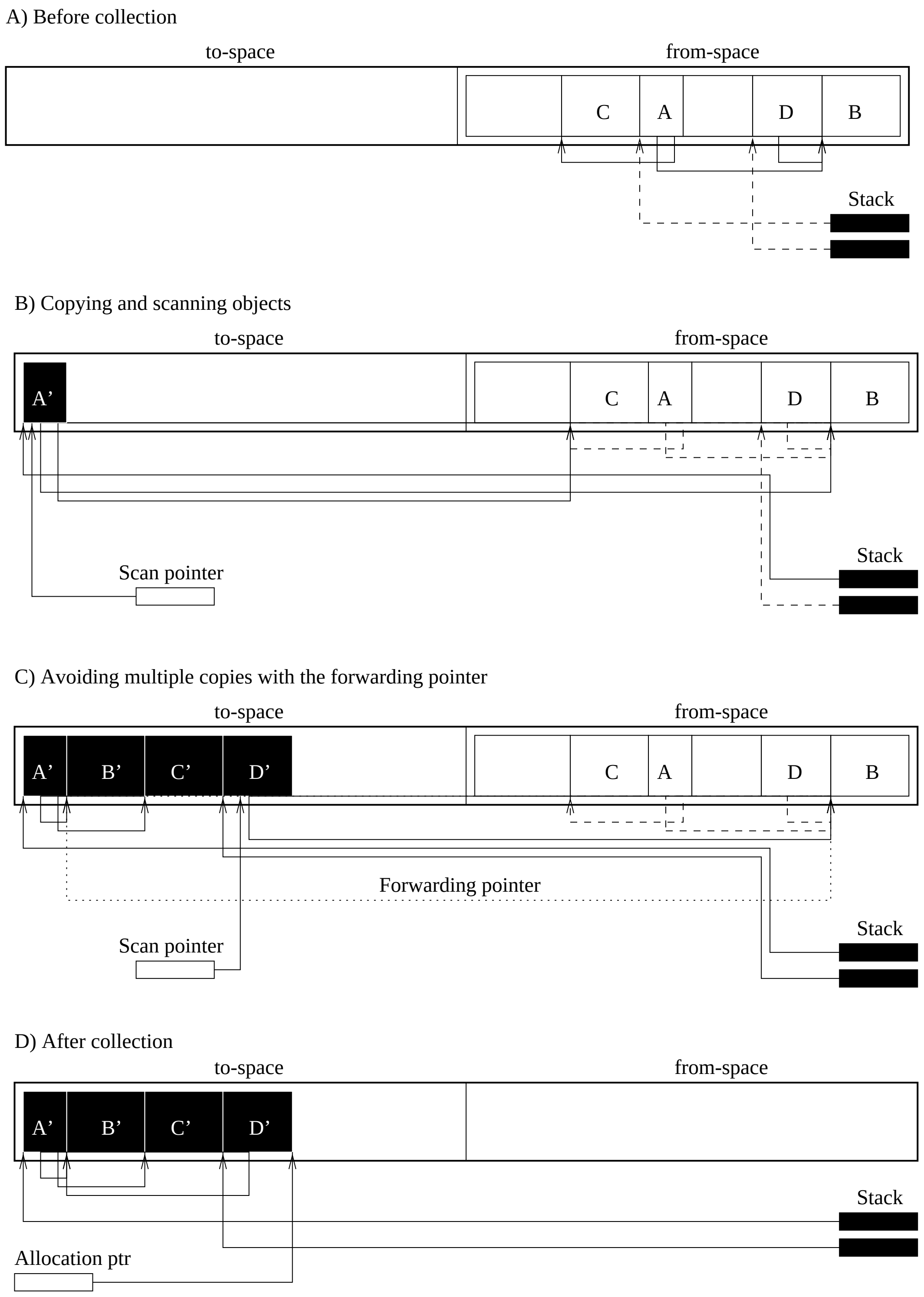

Figure 1.2: *The copy collector.*

the frequency of the garbage collection is to increase the size of the heap. Large, long lived objects, however, will be copied at every collection. Also, only one of the two semi-spaces is used which results in a memory usage of less than fifty percent.

Copy collectors move all the live data out of one region of the memory into another region. The last region is known to contain only garbage and can be reclaimed without further examination. The amount of work that needs to be done is proportional to the amount of life objects. The objects do not really have to be in different areas of the memory to apply this strategy. They can be thought of as part of different *sets*. Initially the *to-set* is empty and all allocated objects are inserted into the *from-set*. During the traversal, the live objects are removed from the from-set and inserted into the to-set. All objects remaining in the from-set are known to be garbage and can be linked into the *free-set*. Instead of using different memory regions, double linked lists can be used. This collection technique is called non-copying implicit collection. With copy collectors it shares the advantage of discarding a sweep phase and only visiting the live object. In addition, it avoids the copying of objects. However, since it performs no compaction, they risk to fragment the heap.

Garbage collecting techniques can be accurate or conservative. They are accurate if they can distinguish pointers from non-pointer data. They are conservative when they only have partial information about the locations of pointers. In that case they must treat every memory word as a potential pointer. They may misidentify words as pointers and wrongly retain garbage. Furthermore, they can not move objects around (as do copy collectors) since this might overwrite data mistakenly interpreted as pointers [PB98]. Conservatism is needed for languages and compilers that do not provide any help for implicit memory management, in particular Pascal, C, and C++. Conservative collectors for C have been proposed. Well known is the collector developed by Boehm [Boe93].

Traditional collectors perform a full scale garbage collection when the application runs out of memory. During this period the application is interrupted. This reduces the responsiveness of a system. In the next section we consider incremental collectors. They improve the responsiveness thru a stepwise collection.

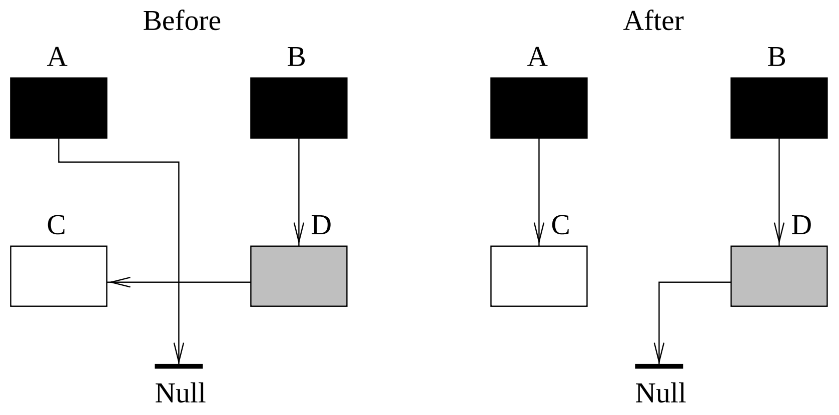

Figure 1.3: *The figure shows a violation of the color invariant after the mutator stored a pointer to C into A, and erased the reference from D to C. If the collector is not protected against modifications by the mutator, object C in the figure will be collected and the data structures will be left in an corrupted state.*

1.2.2 Incremental collectors

Incremental collectors interleave small units of garbage collection with small units of program execution. The garbage collection is not performed as one atomic operation while the application is halted. However, the scanning of the root set is normally done as one atomic operation. Incremental copy collectors, for example, start with a *flip* operation during which the root set is copied into the to-space [Bak78].

The main problem with incremental collectors is that while the collector is tracing out the graph of reachable objects the running application can mutate the graph while the collector is not looking. The running application is therefore called the *mutator*. An incremental collector must be able to keep track of the changes made to the graph of reachable objects and retrace some if necessary. Collectors that move objects such as the copying collector must also protect the mutator. Which one of the two, the application or the collector, is regarded as a mutator depends on who's side you stand. The two can be regarded as two processes that share changing data while maintaining some kind of consistent view.

The abstraction of tricolor marking is helpful in understanding incremental garbage collection. Garbage detection algorithms can be described as a process of traversing the graph of reachable objects and coloring them. The objects subject to collection are conceptually colored white, and by the end of the collection the objects that must be retained must be colored black. The garbage detection phase ends when all reachable objects have been traversed and there are no more objects to blacken. In the mark-and-sweep algorithm marked objects are considered black, unreached objects are white. In the copying algorithm the objects in the to-space are black, the unreached objects in the from-space are white (see Fig. 1.1 and 1.2). To understand the interactions between an incremental collector and the mutator it is helpful to introduce a third color, gray, to signify that an object has been reached but that its descendants might not have been. The gray colored objects form a wave front separating the white from the black objects. No direct reference from the black to the white objects should exist. This property is referred to as the color invariant. The mutator moves pointers around and may the gray wave front (Figure 1.3). The mutator corrupts the collectors tracing when 1) it writes a pointer to a white object into a black object, *and* 2) when it erased the original pointer before the collector sees it. The mutator must prevent the collector somehow when the assumption of the color invariant is violated. There are two basic techniques for the coordination between the two: read barriers and write barriers. Read barriers catch every attempt to read a pointer to a white object. The white object is immediately colored gray. Since the mutator can never access a white object, no pointers to white objects can be stored in black objects. Write barriers catch any attempt to write a pointer into another object. Write barriers come in two flavors. One strategy, *snapshot-at-beginning*, is to ensure that objects never get lost by preventing pointers to get lost. When a path to an object is changed, a new path is created to ensure that the object will be found. One implementation of a write barrier is to store the original pointer onto a stack for later examination before it is overwritten with a new value. Snapshot-at-beginning techniques retain all the objects that were alive at the beginning of the collection. Another strategy, *incremental update* collectors, focuses on the writing of pointers into objects that already have been examined by the collector. The collector's view of the reachable data structures is incrementally updated as changes to the object graph are made. Conceptually, whenever a pointer to a white object is written into a black object, the black object or the white object is reverted gray. Incremental update strategies are able to reclaim objects that become garbage during the collection.

The mutator also has to be protected from changes made by the collector. An incremental copying collector must prevent the mutator to access an old copy in the from-heap instead of the new copy in the to-heap. Copying collectors use read barriers to assure the mutator finds the object at the correct location.

Read barriers and write barriers are conceptually synchronization mechanisms. Before the mutator can access or write a location, the collector must perform some bookkeeping. This bookkeeping, in general, requires only a few actions. These operations can be inlined into the mutators code for efficiency. Since write operations occur less frequent than read operations, write barriers are potentially much cheaper.

1.2.3 Generational collectors

Generational collectors try to benefit from the empirically observed property that most allocated objects live for a very short time, while a small percentage of them live much longer. If an object survives one collection, chances are it will survive many collections. If the collector copies objects or compacts the heap, these long lived objects are copied over and over again. Generational techniques split the heap into several sub-heaps and segregates the objects in the sub-heaps according to their age. New objects are allocated in the first generation sub-heap. When the application runs out of memory, only the first generation sub-heap is scanned. The objects that survive one or more collections are moved to the second generation heap. If most objects are indeed short-lived, most of the first generation heap is freed and few objects need copying to the second generation heap. The older generation sub-heaps are scanned less frequently. Since smaller fragments of the heap are scanned and proportionally more storage space is recovered, the overall efficiency of the garbage collector improves.

1.2.4 Dynamic memory management in real-time applications

Real-time applications must be guaranteed sufficient storage space to execute their task. Real-time application design carefully calculates and allocates all storage space statically before execution. Traditional real-time applications, therefore, do not manage their memory usage dynamically. However, when the application deals with problems that are unpredictable in size and complexity, dynamic memory management is unavoidable. The real-time application designer will therefore need to apply real-time design techniques to the memory allocation and deallocation. This implies a number of constraints:

- The use of CPU and memory of the allocation and deallocation technique must be efficient.
- The time required to allocate and deallocate memory must be tightly bound.
- The time required to read/write a heap allocated object must be tightly bound.
- The application must make sufficient progress and not be interrupted by the collector for too long or too frequent.
- The availability of storage space must be guaranteed.

The efficiency of the CPU and memory usage of the allocation and deallocation technique is not, strictly speaking, a real-time constraint, but can be decisive in the question whether dynamic memory management is possible or not. The CPU efficiency of dynamic memory management in general greatly depends upon the type of the application. Furthermore, the efficiency of a particular garbage collection technique depends on the targeted platform: CPU-type, amount of available memory, memory-management hardware support, data caches, etc. When comparing explicit and implicit memory management techniques, both seem to be comparably efficient for the CPU usage, although techniques relying on garbage collection require more memory. In an early article, Appel even reported how implicit memory techniques can be faster than traditional techniques [App87]. He comes to this conclusion by making the heap arbitrarily big. Zorn tested six existing, large C programs with four explicit and one implicit memory strategy and concludes that CPU overhead of storage managements is application dependent and that garbage collection compares well with other explicit storage management techniques [Zor93]. The memory usage is another important factor, in particular for embedded systems with limited hardware resources [PB98]. On this point garbage collectors score low. Copy collectors have a memory usage of less than fifty percent and are for this reason alone ruled out for memory scarce systems.

One of the major problems with dynamic memory management for real-time applications is that the time needed to allocate and deallocate memory is unpredictable or has unacceptable worst case values. Nilsen examined the average and worst case delays needed for well-known memory allocation algorithms of several operating systems. The conclusion is clear: dynamic memory allocation, independent of garbage collected or not, represent an potential source for unpredictable timing performance [NG95]. On this point, implicit memory management techniques can offer better performance than traditional allocation techniques. Copy collectors only require the incrementing of a pointer plus a boundary test for the allocation, and deallocation comes for free.

Incremental garbage collection techniques require the synchronization between the collector and mutator. Read or write barriers may include extra instructions for every referencing of a heap object. A range check may be required for every pointer dereferencing. This introduces an extra overhead. Also virtual memory protection can be configured to generate page faults on those pages that require synchronization [AEL88]. This solution offers unacceptable bad worst-case times [WJ93, Nil94]. Incremental copy collectors move objects whenever the mutator references an object in the from-space. Baker's real-time copy collector was designed for Lisp and only handles objects of the same size: two words [Bak78]. It therefore offers predictable performance. However, copying during dereferencing is unacceptable if the objects can be arbitrarily large. Nilsen remedies this problem with a lazy copying: the collector only reserves the space for the copied object and keeps the copying for a later date. In general, though, synchronization reduces the system throughput and in many cases offers bad worst case delays. Wilson & Johnstone's non-copying implicit collector uses a write barrier to synchronize the collector and the mutator, making the coordination costs cheaper and more predictable.

The garbage collection can itself be considered as a real-time task with a fixed upper time limit and a fixed period. There are, however, some atomic operations which the garbage collector must do. The delay caused by these operation might determine the limitations on the real-time guarantees. One such atomic operation is the root set scanning. Copy collectors must perform a flip operation. This may require the copying of the root set (unless objects are copied lazily). This flip operation must also invalidate the memory caches to assure that all stored data is written thru. Another concern is that the application must be guaranteed sufficient progress: even if the interruption of the collector is small and bounded in time, they must not occur too often. The collector must guarantee that for any given increment of computation, a minimum amount of the CPU is always available for the running application [WJ93]. Baker's algorithm, for example, closely integrated program execution and collector actions. The collector's interruptions may cluster together, preventing the application to meet its deadlines.

One of the reasons why real-time developers are reluctant to use dynamic memory management, is that they fear that the heap will get so fragmented that future storage request can not be satisfied. They do not use dynamic memory allocation at all, or use predictable allocation techniques that use a number of fixed size allocation cells to prevent fragmentation at the expense of storage overhead (for example, Kingsley's powers-of-two buddy algorithm [Kin82]). Wilson & Johnstone limit the risk of fragmentation of their real-time non-copying implicit collector by rounding the size of storage space requests up to a power of two, much like the buddy allocation algorithm [WJ93]. Another issue of real-time garbage collection is the guarantee of sufficient progress. The garbage collector has a real-time task of its own: collect the heap before all storage space is allocated. The garbage collection must therefore be able to keep up with the mutator's allocation rate.

Nilsen's study of available techniques [Nil94] concludes that without the assistance of specialized hardware, the existing real-time garbage collection techniques may generalize to soft real-time applications. According to him real-time system designers argue that these techniques do not provide the fine granularity of timing behavior that is required for the creation of reliable real-time systems. Wilson & Johnstone argue that their incremental non-copying implicit garbage collector satisfies hard real-time constraints. We have found no studies that test their claims in fine grained real-time applications. As discussed in section 1.1, real-time design techniques measure the maximum time needed for every instruction. Wilson remarks that this approach unrealistically emphasizes the smallest operations [Wil94]. For some complex tasks thousand of instruction are executed. For most applications, a more realistic requirement for real-time performance is that the application always be able to use the CPU for a given fraction of time at a time scale relevant to the application. Many real-time applications can function correctly if the garbage collector sometimes stops the application, provided that these pauses are not to frequent and too short to be relevant to the applications deadlines.

This overview of garbage collection techniques may give the reader an idea of the complexity of the issue. Real-time garbage collections remains a topic of discussion. Many authors propose real-time garbage collection techniques, but there claims often seem to be valid for particular applications, not for the general case or for heavy duty or short delay real-time systems. We have not found any article that clearly states that it is possible or impossible to implement fine grained real-time garbage collection on commercial uniprocessor systems.

1.3 Multi-tasking

1.3.1 Processes

Uniprocessor computers can only accomplish one task at a time. To simulate concurrency, multitasking techniques are used. Time-sharing systems allowing the users simultaneous access have been around for a long time. The standard Unix term for an active task in an operating system is a *process.* A process can be thought of as a computer program in action. Associated with a process are a number of system resources, such as a memory address space, files, and other system resources. The process also includes a program counter, a stack pointer, a set of registers. Many process can be running concurrently, each process in its own protected virtual address space. Limited communication between process is possible thru secure kernel mechanism (signals, sockets, shared memory). This separation of processes must guarantee that one process cannot corrupt another process (Fig. 1.4).

The operating system is able to emulate multiple simultaneous tasks by giving each process a chance to run on the CPU for a limited amount of time. Intelligent scheduling of the processes can give the user the impression of truly simultaneous activities. There are many occasions when a process does not require the CPU, for example, when the program is waiting for user input. While the process waits for input/output, the system gives the CPU to other deserving processes. The suspending of one process and the resuming of another is called a *process context*

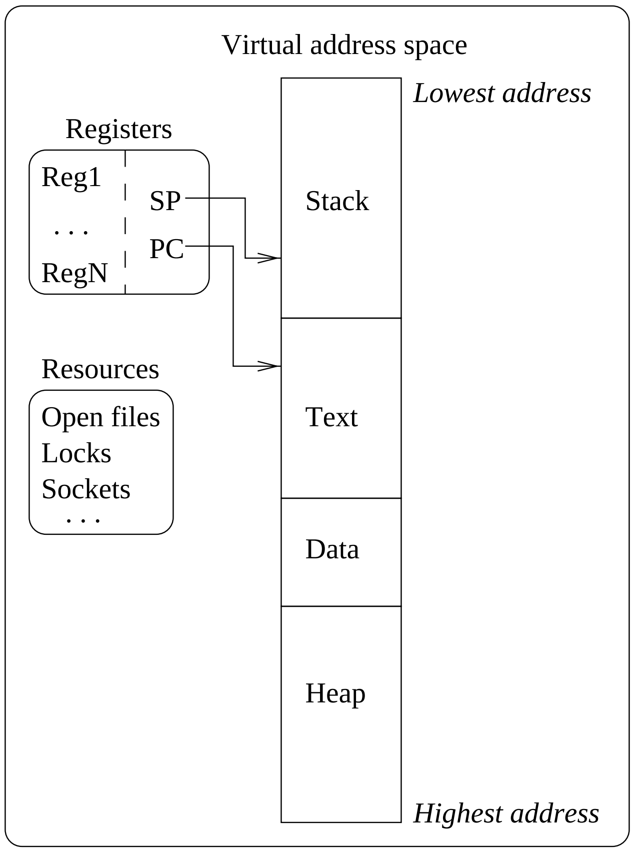

Figure 1.4: *The resources, registers, and virtual address space of a process.*

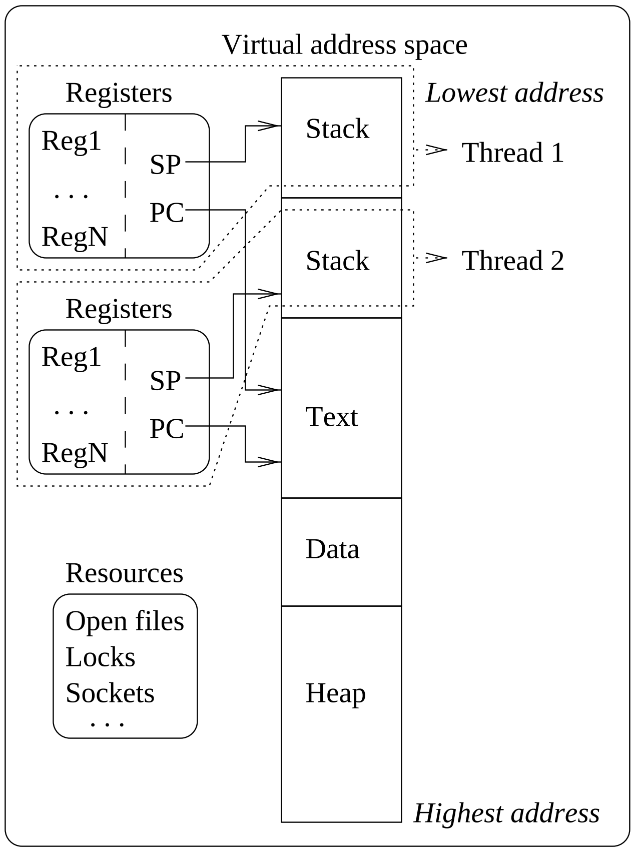

Figure 1.5: *The structure of the threads inside a process.*

switch. The system saves the registers, the program counter, the stack pointer of the suspended process and re-installs those of another, runnable process. What process is scheduled next is determined by the scheduling algorithm.

1.3.2 Threads

A process executes one program. Processes run in their own address space and have their own resources. Communication between processes is limited, mainly to prevent one process from corrupting another. In many cases concurrency within a given program can be desirable. The old way of designing multiple tasks within a process was to fork of a child process to perform a subtask. In the the eighties, the general operating system community embarked upon several research efforts focused on *threaded* program designs, as typified by the Mach OS developers at Carnegie-Mellon [Loe92]. With the dawn of the nineties, threads became established in various UNIX operating systems.

The thread model takes a process and divides it into two parts. The first part contains resources used across the whole program, such as program instructions, global data, the virtual memory space, and file descriptors. This part is still referred to as the process. The second part contains information related to the execution state, such as program counter and a stack. This part is referred to as a thread [NBPF96]. A process can integrate several threads (Fig. 1.5). Threads are the units of execution; they act as points of control within a process. The process provides the contained threads with an address space and other resources.

Not all operating systems provide threads. For those systems that do not provide threads it is possible to use a thread library. Languages implementations that define threads as a language constructs generally do not rely on threads provided by the operating system but provide their own thread system. The overwhelming reason to use threads over multi-processes lies in the following benefit: threads require less program and system overhead to run than processes do. This translates into a performance gain. We give a concrete example taken from the Real-Time Encyclopedia[1]: on the IRIX systems developed by Silicon Graphics Inc. a process context switch takes 50 μsec, while a thread context switch takes 6 μsec. Another advantage is that all threads share the resources of the process in which they exist. Multiple processes can use shared memory if specially set up. However, communication between processes involves a call to the system and is more expensive than communication between threads. Communication between threads can usually be done in the user address space.

The operating system continuously selects a single thread to run from a systemwide collection of all threads that are not waiting for the completion of some I/O or are blocked by some other activity. It is beneficial to give those threads that perform important task advantage over those that do background work. The choice of any given thread for special scheduling treatment is determined by the setting of the thread's scheduling priority, scheduling policy, and scheduling scope. When the system needs to determine what thread will run, it should find out which thread needs it most. The threads priority determines which thread, in relation to the other threads, get preferential access to the CPU. In general, the system holds an array of priority cues. When looking for a runnable thread the system starts looking in the highest priority queue and works its way down to

[1]http://www.realtime-info.be/encyc

the lower priority queues to find the first thread. When and how long the thread runs is determined by the scheduling policy. Well known scheduling policies are first-in-first-out (FIFO), round robin, and time sharing with priority adjustments. FIFO picks the first thread out of the queue and lets it run till it blocks on a I/O operation. The blocked thread is then inserted at the end of the queue. The round robin policy lets the thread run for fixed amount of time, called a quantum. If the thread does not block before the quantum, the scheduler blocks the thread and places it at the end of the queue. The time sharing with priority adjustments will increase the threads priority if it blocks before the quantum expires. Threads blocking on I/O are given privileges over CPU consuming threads using all their quantum.

Furthermore, threads can be scheduled within system scope or within process scope. When scheduling occurs within system scope, the thread competes against all runnable processes and threads on the system level. The system scheduler determines which threat or process will run. When scheduling occurs in process scope, the thread competes with the other threads within the program. The system scheduler determines which process will run, the process scheduler determines which thread will run. Some systems allow the developer to specify in what scope the thread will be scheduled. However, in many cases threading is provided by a threads library in which case threads are always scheduled within process scope.

Access to the shared resources must be synchronized. If uncontrolled access was allowed race conditions may occur: two threads access the resource at the same time leaving the program in a corrupted state. There exist several techniques to assure the synchronization between threads: semaphores, mutex variables, locks, condition variables, and monitors. For this text we will only use the notion of *critical zones*. A critical zone is a piece of code that must be executed atomically. This means that only one process can be inside a critical zone at a time. When another process tries to enter a critical zone, it looses the CPU and is put in a waiting line until the critical zone is deserted.

1.3.3 Real-time tasks

A real-time task responds to unpredictable external events in a timely predictable way. After an event occurred an action has to be taken within a predetermined time limit. Missing a deadline is considered a severe software fault. A real-time task must be sure it can acquire the CPU when it needs to, and that it can complete the job within time delay. Since several real-time tasks might be running in the system concurrently, conflicts may arise. Since real-time tasks require the cooperation from the system we will discuss briefly real-time operating systems (RTOS). According to [TM97] for a operating system to be labeled real-time, the following requirements should be met:

- The RTOS has to be multi-threaded and preemptible. To achieve this the scheduler should be able to preempt any thread in the system and give the resource to the thread that needs it most. The Linux system, for example, can not be preempted: all kernel threads run till completion even if real-time threads are waiting.
- The notion of thread priority has to exist. In an ideal situation, a RTOS gives the resources to the thread that has the closest deadline to meet. The system must know which thread has to finish its job and how much time the job will take. This is far too difficult to implement and OS developers therefore introduced the concept of priority levels.
- The RTOS has to support predictable thread synchronization mechanisms and a system of priority inheritance has to exist. The combination of thread priority and resource sharing leads to the classical problem of priority inversion. The generally accepted solution is priority inheritance. We will discuss this below.
- The OS behavior should be known. The time needed for interrupt handling, context switching, memory access, or acquiring a lock should be documented. Since real-time applications designers measure worst case latencies, these figures should be available to the developers.

The Real-Time Encyclopedia provides a long list of available real-time, mostly embedded systems. The best known public research project on kernel design and real-time systems is the Mach kernel with the real-time extensions (RT-Mach, [TNR90]). The POSIX thread proposal defines an extension for real-time threads [POS96]. The POSIX real-time thread extensions are optional, but several operating systems provide them.

Threads and processes are attributed a priority. This priority is used in the scheduling algorithm to determine what thread or process will run next. Priority inversion can occur in the following conditions 1) there are three threads, one with a low, one with a medium, and one with a high priority, 2) the low and the high priority thread share a resource that is protected by a synchronization mechanism. Consider the situation where the low priority thread acquires exclusive access to the shared resource. The medium priority threads wakes up and preempts the low priority thread. When the high priority thread is resumed and tries to acquire access to the resource, it is blocked till the low priority thread releases the shared resource. Since the low priority thread has been suspended by the medium priority thread, the high priority thread is forced to wait for the medium priority thread and could be delayed for an indefinite period of time. The generally accepted solution to the problem is the technique of priority inheritance [SRL87]. Priority inheritance is an attribute of the synchronization mechanism (lock, semaphore, mutex, etc.). Conceptually, it boosts the priority of a low priority thread that acquired a shared resource whenever a high priority thread is waiting for that resource. The priority of the low level thread is set to the priority of the high level thread and can run un-interrupted by medium priority threads till it relinquishes the shared resource. In this way, the high priority thread's worst case blocking time is bounded by the size of the critical zone.

In applications where the workload consists of a set of periodic tasks each with fixed length execution times, the rate monotonic scheduling algorithm, developed by Liu & Layland, can guarantee schedulability. The rate monotonic algorithm simply says that the more frequently a task runs (the higher its frequency), the higher its priority should be. If an application naturally is completely periodic or can be made completely periodic, then the developer is in luck because the rate monotonic scheduling algorithm can be employed and a correct assertion can be made about the application meeting its deadlines. Rate monotonic scheduling statically analyses the requirements, which means that the type and number of tasks are known before hand and that their priorities are fixed.

The Real-Time Mach kernel uses a time-driven scheduler based on a rate monotonic scheduling paradigm. A thread can be defined for a real-time or a non-real-time activity. A real-time thread can also be defined as a hard or a soft real-time thread, and as periodic or aperiodic based on the nature of its activity. The primary attributes of a periodic thread are its worst case execution time and its period. Aperiodic threads are generally resumed as a result of external events. Their main attributes are the worst case execution time, the worst case interval between two events, and their deadline.

The real-time processing of digital video and audio in multimedia applications requires rigid latency and throughput. These applications use several system resources simultaneously such as CPU time, memory, disk and network bandwidth. In addition, the work load of these applications may vary according to user input and new tasks may be added or removed to the system dynamically. In order to provide acceptable output, a resource management is necessary that exceeds the classical scheduling algorithm problem. Furthermore, it should be left to the developer and end user to define the measures of goodness for the performance of the application. Ideally one would provide a range of real-time services from "guaranteed response time" to "best effort" and allow developers and users to decide whether the extra cost of a more rigid real-time service is worth the perceived benefit. The problem is best expressed in terms of real-time resource allocation and control. A more recent notion in real-time systems is *quality of service*. Quality of service (QoS) is used in networking, multimedia, distributed, and real-time systems. Applications contending for system resources must satisfy timing, reliability, and security constraints, as well as applications specific quality requirements. The problem is expressed in terms of allocating sufficient resources to different applications in order to satisfy their quality of service requirements. Requirements such as timeliness and reliable data delivery have to be traded of against each other. Applications may content for network bandwidth; real-time applications may need to have simultaneous access to memory cycles or disk bandwidth [Jef92, CSSL97, RLLS97]. The RT-Mach kernel uses a QoS-manager to allocate resources to an application. Each application can operate at any resource allocation point within minimum and maximum thresholds. Different adjustment policies are used to negotiate a particular resource allocation. The RT-Mach QoS server can allow applications to dynamically adjust there resource needs based on system load, user input or application requirements.

Quality of service resource allocation supposes that an application may need to satisfy many requirements and require access to many resources. Furthermore, the application requires a minimum resource allocation to perform acceptably. The application utility is defined loosely as the increase in performance when when the application is allocated a new share in the resources. Every application is given a weight, such that the total utility of the system can be measured as the sum of the weighted utilities. Consider, for example, an audio application outputting its samples over the network at a rate of 8kHz. Delivering this quality requires a certain amount of CPU time and network bandwidth. If the application increases its sampling rate to 16kHz the CPU time and bandwidth may double. However, the increase in sound quality might justify the extra claim on the resources: the utility is high. If the application moves the sampling rate to 32kHz, CPU time and bandwidth may double again, but the utility might not be high enough to justify the increase of resource usage. When several applications run concurrently and compete for the resources a negotiation will determine the allocation of the resources between the applications. Quality of service of service not only applies to the usage of the CPU time but includes other system dimensions as well. Before a new task is accepted by the kernel, it must specify its requirements. Once accepted the kernel guarantees that the demand is available.

Chapter 2

An introduction to computer music

Music creation in the western culture has traditionally been a two step process. First, the composer creates a musical score. Then the score is handed over to the musician to interpret the notated music. We will see that in both processes, the computer plays an important role. Sound synthesis techniques allow the creation of new sounds and have been the subject of many research projects. Programs for the specification, control, and execution of synthesis techniques are wide spread. Less common is the research on computer aided music composition. In this domain, tools, data structures, and interfaces are defined to help to composer to represent, organize, and explore musical ideas. Although, with the use of computer, the gap between musical writing and sonic realization can be bridged conceptually, most existing environments for computer music maintain the separation between musical writing and performance. It is our believe that this separation greatly prevents exploring the possibilities offered by the computer. We are interested in the fine tuning of the relations, algorithms, and parameters that describe the organization of a music piece on every possible level of abstraction. In this chapter we give a brief overview of the concepts, techniques, and programs found in the field of computer music.

2.1 Sound synthesis

2.1.1 Digitizing sound

An acoustical signal $p(t)$ can be converted to an electrical signal $e(t)$ and vice versa using microphones and loudspeakers. The electrical signal $e(t)$ can be represented in a digital format as a series of digitally coded integers $s(n)$. To convert $e(t)$ to a digital signal a two step quantification is necessary. First, the signal will be sampled at fixed time intervals T_s. The value of the sampling frequency $F_s = 1/T_s$ is determined by the highest frequency component F_m present in the signal. According to the Nyquist theorem the sampling frequency should be at least twice the maximum frequency contained in the signal: $F_s > 2F_m$. Second, the values $e(nT_s)$ must be quantized to fit an integer value. The simple method would be to round the values $e(nT_s)$ to the nearest integer value to obtain the series s(n). The digitizing is realized by a analog-to-digital converter (ADC). With a digital-to-analog converter (DAC) the digital signal $s(n)$ can be changed back to an electrical signal and fed to a loudspeaker [Mat69].

This correspondence between series of numbers and sounds allows us to use digital methods and devices for the synthesis and manipulation of sounds. Instead of using existing, digitized sounds, we can apply digital synthesis techniques to create new, unexisting sounds artificially. A whole new field of undiscovered sounds opens up. The sound samples can be calculated out of time, i.e. the sound is calculated in its whole length and is played when the synthesis is finished. Or the sound can be generated in real-time: the synthesis and the playing are interleaved such that the sound can be heard at the time of the calculation.

More than one sound signal can be generated at the same time. If a stereo signal is desired, two signals must be synthesized. In the general case N signals can be generated. Each signal is sometimes called a channel or a track. The convenient and most used way to stock a multi-channel sound is to interleave the samples of the distinct channels. We will call a *frame* the set of N samples $f = (s_0(n), \ldots, s_{N-1}(n))$ of the N channels signal.

Since any audible sound can be represented as a series of sampled values, theoretically any sound can be generated by the computer. The digital representation of a sound, however, asks for an enormous amount of data and it is therefore inconceivable to generate this data by hand. To describe and facilitate the generation of sound *synthesis models* are used. These models take a number of *control parameters* and produce a digitized sound signal.

Two main categories of models can be distinguished. The first category models the sound signal and comes historically first. This category is generally called signal modeling. Signal models describe the acoustical structure of the sound. The desired waveform is described using signal processing techniques such as oscillators, filters, and amplifiers.

The second category models the acoustical causes of sound generation. This category is generally called physical modeling. Physical models depart from a description of the mechanical causes that produce the sound. These models describe the sound production mechanisms and not the resulting waveform [Ris93], [DP93]. There are different techniques to translate physical descriptions of music instruments into a digital synthesis technique. Three main techniques can be distinguished: the wave-guide models [Smi96, Smi97], the modal descriptions of instruments [MA93, EIC95], and the mechanical models [CLF93]. In this text we will only consider the signal models.

2.1.2 Signal models

Among the signal models some of the well known techniques are [DP93, Roa96]:

- Sampling,
- Wave table synthesis,
- Additive synthesis,
- Phase vocoder,
- Granular synthesis,
- Formantic waveform synthesis,
- Subtractive synthesis,
- Wave shaping,
- Frequency modulation,

The sampling technique stores concrete sounds, often recordings of musical instruments, into tables. The synthesis consists of reproducing the sound by reading the values in the table. The frequency of the sound can be modified by changing the speed at which the values in the table are read, and an amplitude curve can be applied to change the dynamics and duration. The technique is very simple but effective to produce rich sounds quickly.

Like sampling, wave table synthesis reads the values stocked in a table to produce the sound. The table, however, contains only one period of the wave form of the signal. It is a fast technique to implements sine-wave, rectangular, or triangular wave oscillators [Mat69]. The waveform stocked in the table can have any shape, and may vary during the synthesis.

In additive synthesis, complex sounds are produced by the superposition of elementary sounds. The goal is for the constituent sounds to fuse together, and the result to be perceived as a single sound. Any almost periodic sound can be approximated as a sum of sinusoids, each sinusoid controlled in frequency and amplitude. Additive synthesis provides great generality, allowing to produce almost any sound. But a problem arises because of the large amount of data to be specified for each note: two control functions must be specified for each component. The necessary data can be deduced from frequency analysis techniques such as the Fourier transform.

The phase vocoder is an analysis/synthesis technique based upon the Fourier transform. It is similar to additive synthesis, however, the phase vocoder does not require any hypothesis on the analyzed signal, apart from its slow variation. Both additive synthesis and the phase vocoder split amplitude and frequency information in time. This separation allows for such transformations as changing the duration of the sound without changing its frequency contents, or transposing the signal but keeping its duration constant [Moo78].

Granular synthesis starts from the idea of dividing the sound in time into a sequence of simple elements called grains. The parameters of this technique are the waveform of the grain, its duration, and its temporal location. The first type of granular synthesis consists of using, as grain waveform, the waveform of real sound segments and then using the grain waveforms in synthesis in a different order or at various times. This method refers back to the synthesis of sampled sounds, except that in this case the elements are no longer complete sounds but their fragments. The second type consists of using waveforms such as Gauss functions modulated in frequency, which have the property of locating the energy in the time-frequency plane.

Formantic waveform synthesis (abbreviation FOF, from french *Forme d'Onde Formantique*) can be considered a granular synthesis technique. The grain waveforms are sinusoids with decreasing exponential envelope. This waveform approximates the impulse response of a second order filter. It has a formantic spectral envelope. The overall effect is obtained by the presence of various waveforms of this type. The waveforms are repeated, synchronous to the pitch of the desired sound. By varying the repetition period, the frequency of the sound varies, whereas by varying the basic waveform, the spectral envelope varies. With the use of FOF generators in parallel one can easily describe arbitrary time-varying spectra in ways that are at once simpler and more stable then the equivalent second-order filter bank [RPB93].

Many synthesis techniques are based on the transformation of real signals or signals that have been generated using one of the above mentioned methods. A first set of transformations consists of linear transformations. These transformations are based mainly on the use of digital filters. According to the frequency response of the filter we can vary the general trend of the spectrum of the input signal. Thus, the output will combine temporal variations of the input and the filter. When a particular rich signal is used and the transform function of the filter has a very specific shape, this method of generating sound is usually called subtractive synthesis or source/filter synthesis. The common procedure is linear prediction, which employs an impulse source or noise and a recursive filter [Mak75]. A closely related technique is cross synthesis in which the spectral evolution of one sound is imposed upon a second sound. In the field of linear transformations we also find delay lines and comb filters. With the combination of delay lines and digital filters a reverberating effect can be obtained.

The technique of wave shaping is probably the most applied non-linear transformation. A linear filter can change the amplitude and phase of a sinusoid, but not its waveform, whereas the aim of wave shaping is to change the waveform. Wave shaping distorts the signal introducing new harmonics, but keeps the period of the signal unchanged. Wave shaping exploits this property to generate signals, rich in harmonics, from a simple sinusoid. If the function $F(x)$ describes the distortion, the input $x(n) = \sin 2\pi fn$ is converted into $y(n) = F(x(n)) = F(\sin 2\pi fn)$.

Frequency modulation (FM) has become one of the most widely used synthesis techniques. The technique consists of modulating the instantaneous frequency of a sinusoidal carrier according to the behavior of another signal (modulator), which is usually sinusoidal. Several variants exist using complex modulators or modulators in cascade.

2.1.3 Synthesis applications

Signal models try to capture the characteristics of the sound they intend to produce. These characteristics describe the acoustical or physical qualities of the sound. Many of the signal synthesis models can be described in terms of basic units, such as oscillators, filters, delay lines, put in cascade or in parallel. These basic modules are often called *unit-generators*, a name due to Max Mathews. He developed the first suite of synthesis programs, named Music I to Music V [Mat69], [MMR74]. This early work spawned numerous descendants. The term Music N is generally used to refer to this extended family of programs.

In Music N unit-generators such as oscillators and random generators generate streams of audio samples, while signal modifiers such as filters, amplifiers, and modulators process these streams. Networks of these unit-generators can be patched together. The resulting network is called an *instrument*. The action of the instrument is defined by the connectivity graph of the unit-generators as well as by the acoustical parameters that control the action of the unit-generators (parameters such as frequency and amplitude for oscillators, for example). The activation of instruments is controlled by *note statements* that bind the instrument, the action times, and the acoustical parameters together. A *score* is a collection of note statements and instrument definitions [Loy89].

Music N creates instrument templates, instantiates them at the right times, and binds the acoustical parameters to the unit-generators. It also merges the outputs of the various instruments into the final output. This output is stored into a file or sent to the sound output device. In Music V several instances of an instruments can exist simultaneously. Music V also generates a block of samples for each unit-generator on each pass, instead of one sample. This largely increases the efficiency.

Two other implementations of the Music N model are *cmusic*, written by F.R. Moore, and *Csound* , written by Barry Vercoe [Ver86]. Because both are written in C, these implementations are available on a wide range of platforms.

The approach of signal processing networks is found in most customizable synthesis environments, such as SynthBuilder [PSS$^+$98], and Max/FTS [Puc91b, Puc91a]. Both provide a graphical environment to construct patches. The MAX/FTS environment uses a message passing approach to communicate between the signal and control objects. Complex patches can be created that describe the synthesis and the control of synthesis. A patch can be in two states: in the first state the patch can be edited, in the second state the patch is executed. The environment is extendible and programmers can write external objects. Both of the above mentioned systems are designed for real-time performance.

The CHANT program, developed by Xavier Rodet and his colleagues, offers a model of vocal synthesis based on the formantic waveform synthesis, explained above. Apart from a new synthesis technique they worked on a better control over it. They observed that the Music N patch model had its limitations: "patch languages that exists are weak in their ability to specify the elaborate control levels that resemble interpretation by an instrumentalist, for example, expressiveness, context-dependent decisions, timbre quality, intonation, stress and nuances" [RPB93]. To overcome this they adopted a hierarchical synthesis-by-rule methodology. The method of controlling the collection of FOF generators is thru a large set of global parameters that can be changed by the concurrent execution of cooperating subroutines that implement the synthesis rules corresponding to the vocal model under development. All the parameters begin with default values, which, when executed, produce a normalized vocal synthesis. Libraries of previously developed routines that implement various rule sets are available for composers; as well, there are facilities to create new rule sets and to modify and extend existing ones. These libraries are called knowledge models of different productions. It became clear that, as the cooperating rule sets grew and became more complicated, that it was turning more and more into an AI problem to represent the control flow. The FORMES language, based on Lisp, was developed for this purpose, and will be described later.

Foo is a music composition environment developed and designed by Eckel & González-Arroyo [EGA94]. Their research focused on composed sound which can be fully integrated in a compositional process. A musical object will in general be a complex object, decomposable into simpler elements, organized under a logical structure. A music piece can be regarded as a dynamic, compound structure, where behavioral laws and signal processing patches combine. This structure can be viewed both as a sound production entity and as a logical object, artistically meaningful. They do not set a priori a boundary between the level of sound object definition and that of its musical manipulation. They wish to embed in one whole environment all actions from the micro-control of the signal processing to the composition of the score and search for new perspectives of relationship between sound matter, musical material and form. Foo consists of two parts: a kernel layer and a control layer. The kernel layer provides the necessary low level abstractions to define and execute signal processing patches and is written in Objective-C. The control layer consists of a set of Scheme types and procedures for the creation, representation, and manipulation of sound concepts and musical objects in general. Foo allows the expression of temporal relationships different modules. The two most important parameters of this time context are the time origin and the duration.

2.2 Computer aided composition

The western culture has always maintained a close relationship between music and numbers [AC95]. Music analysts admit that most composers use both calculations and intuition in variable proportions. Until the eighteenth century, music, as a science of numbers, constitutes a field of theoretical speculation and practical experimentation for a great number of intellectuals, may they be composers or not. Mersenne in his treatise *Harmonie Universelle* poses the question if it is imaginable to compose the best chant possible. The answer is negative: the number of possible chants is too big, the composer can only proceed using trial and error. Developing the idea of musical combinations further in the seventeenth and eighteenth century, lead to the idea of automating certain aspects of the musical composition, to the fabrication of the first *music machines* (*arca musurgia* by the German Athanasius Kircher around 1650), and to musical games, such as the *Musikalisches Wurfespiel* by Mozart.

As opposed to the generation and processing of audio signal by the means of digital signal processing, or to the instrument-level computerized interaction by the use of Musical Instrument Digital Interface (MIDI, [Loy85]), systems for computer aided composition (CAC) focus on helping the composers (and often the musicologists as well) to formalize and experiment with the deep structures and the dynamics of their musical languages. This approach was envisioned by Ada Lovelace in the end of the 19th century when she realized that the unachieved computing machine designed by Babbage should be able to manipulate symbols as well as numbers, and, by computing relations between symbols, could become someday a composing machine for several artistic disciplines including music. When this prediction became eventually true, in the late fifties, the first computer music applications were essentially formal and algorithmic [Hil70, Xen90, Bar86]. In the following years, the computer music people's interest shifted massively to sound synthesis and processing [Mat69] then to real-time interaction. The "modern" CAC approach, appeared in the eighties, was favored by the progress in computer languages, architectures, and graphical user interfaces and the emergence of personal systems.

The Formes program [RC84], although mainly devoted to the control of sound synthesis, was really a compositional environment, with a high level object oriented architecture.

Crime was written in LeLisp in a Unix environment, and designed by G. Assayag and composer C. Malherbe. The recent availability of the Apple Macintosh and of cheap MIDI systems had become tempting at that time. A series of software prototypes on the Mac, partly derived from the Crime project and expertised by such composers as Tristan Murail and Magnus Lindberg, took a final form with Mikael Laurson's original idea to bring a graphical programming interface to Lisp. This form was the PatchWork environment, by M. Laurson, J. Duthen and C. Rueda [MJ89]. The combination of programming simplicity, highly visual interface and personal computing concept created a real infatuation for PatchWork among European composers with highly diverse musical and aesthetic backgrounds. PatchWork is a visual interface to the Lisp language. As such, it is a purely functional language, except that some functions may retain a local state and provide a graphical editor for inspecting and editing this state.

Developed during the same period by Francis Courtot, CARLA was an attempt to use a visual programming interface to a Prolog-based logic programming system [Cou92a].

OpenMusic, designed by G. Assayag and C. Agon [GCFP97, Ago98], is a visual interface to CLOS, the Common Lisp Object System. Aside from being a superset of PatchWork, it opens new territories by allowing the composer to visually design sophisticated musical object classes. It introduces the Maquette concept which enables high level control of musical material over time and revises the PatchWork visual language in a modern way.

Additional environments for CAC include Loco [DH88], Common Music [Tau91], and Artic/Canon/Nyquist [DR86, Dan89, Dan93] and DMix [Opp96]. We will discuss aspects of these environments in more detail in the next chapter where we will focus on the issues in the organization and manipulation of time.

Chapter 3

Issues in the organization and manipulation of time

Time is without doubt the most important dimension in music. Its representation and manipulation is crucial for the expressiveness of a composition environment, yet has proven to be a formidable problem. Discussions on time can be found in the fields of multimedia authoring, temporal algebra, auditive perception and cognition, musicology, composition, philosophy, and physics. In this section we will limit our discussion mainly to the representation of time found in systems for music and include some issues found in articles on multimedia authoring. We will use the generic term *temporal object* to denote any object, musical or other, that has a location and duration in time. Structures that group several temporal objects we will call *composite (temporal) objects*. When we use musical examples, we may use the terms *sound* or *music objects*.

The following list gives an idea of some of the issues in the discussion of time representation and organization:

- The description of timing information.
- The contruction of composite temporal objects.
- The use of composite objects as basic elements.
- The definition of temporal relations between objects.
- The manipulation and transformation of composite objects.

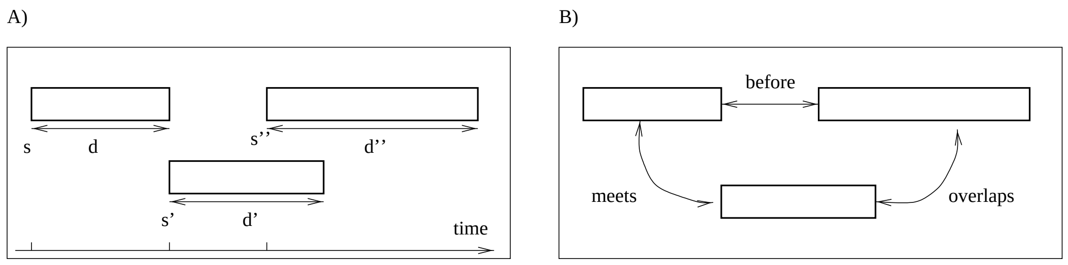

Figure 3.1: *Precise timing versus relations in composite time structures*

- The interactive, dynamic control of a composition.
- The relations between continuous time and event time.
- Time and hierarchy in musical structures.
- The manipulation of continuous time functions.
- The relation between composed time and performance time.
- The representation of temporal information during runtime.
- The relation between "out-of-time" structures and "in-time" realization [Xen90].

A discussion of some time representation issues in music can be found in [DDH97] and [Hon93]. The last article has the additional merit of trying to link discussions in the field of music cognition to representations used in music composition systems. The above issues are so entangled that we can not discuss them separately. We will therefore organize the following discussion in two main parts. In section 3.1, we discuss the structured organization of music pieces. This part handles the structuring of events and durations. In section 3.2 considers the use of continuous time functions in music pieces.

3.1 Structural organization

A piece consists of a set temporal objects. Various approaches exist to organize these objects in time and to describe the relationships among them. Structural organization is essential for any multimedia system; without it, composition is impossible.

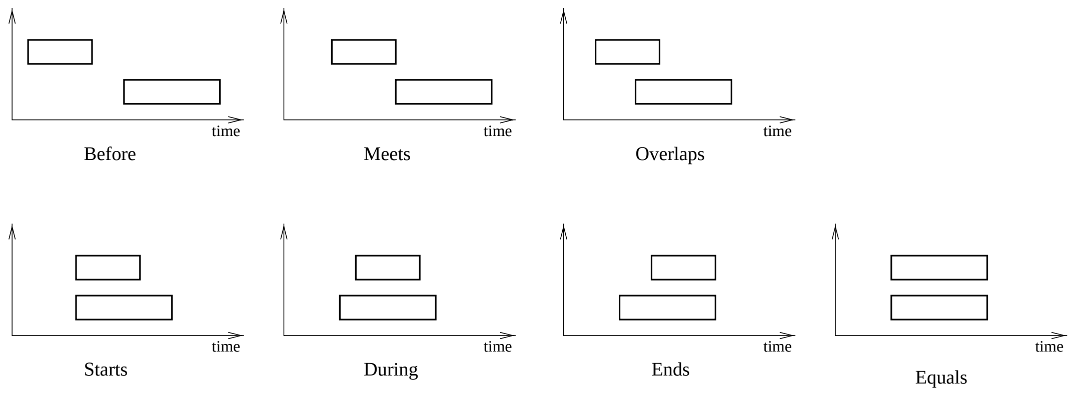

Figure 3.2: *The Allen relations between two intervals*

A first important distinction between several composition systems is the importance given to the specification of the precise start times or to the structural relations in the piece (Fig. 3.1). When a higher importance is given to precise timing, the relations between the objects have to be calculated. When a higher importance is given to the relations, the precise timing of the objects have to be calculated. Honing calls the first approach implicit structuring and the latter explicit structuring [Hon93].

Precise timing is used by most synthesis systems, including Music N and the MIDI file standard. The objects are placed on a *time line* with their precise location and duration. A distinction can be made between absolute and proportional time bases. An absolute time base requires the times to be given in absolute time, i.e. in seconds. In a proportional time base the times are specified in regard to some relative unit, often a note or beat value.

Explicit structuring is often based on the Allen relations. He distinguishes thirteen possible relations between two intervals [BL89]. If we leave out the inverse relations seven remain: *before*, *meets*, *overlaps*, *starts*, *during*, *ends*, and *equals* (Fig. 3.2). The Allen relations have also been studied and extended in the case of cyclic time models [PC98].

Most composition systems use implicit structuring but generally offer Allen-like relations through the parallel and the sequential structuring [DH88, Bal92, Dan89, Cou92b, RC84]. In the parallel structuring, all elements start at the same time; in the sequential structuring every element meets the next element.

The use of composite objects as a basic building block in other composite objects allows the representation of a piece as a directed graph. Parallel and sequential structures can in general be embedded in other structures. This approach is often refered to as the *container-contained* relation. Most synthesis programs, however, only provide a one dimensional time line and no or very limited means to group temporal objects.

The straightforward approach to create composite structures is to place the temporal objects into a container manually. Piano roll and music notation editors are examples of graphical interfaces to construct music pieces manually.

Music is by nature very redundant: the same group of elements are used repeatedly throughout the piece. Buxton and colleagues reduce redundancy in musical structures thru the use of so called instanciation. A reoccuring sequence is only represented once. Several "instances" refers to this prototype at the appropriate places in the score [BRP$^+$78].

The most interesting approach for music is the generation of composite objects thru the use of a functional, algorithmic, or algebraic description. It is a desirable if not necessary feature for composition environments. This approach raises the issue of the language used for this description. Most of the cited music composition languages are extensions to a Lisp-style language [SJ93]. The environments Formes [RC84], Loco [DH88], Common Music [Tau91], and Artic/Canon/Nyquist [DR86, Dan89, Dan93] use a Lisp-like language interpreter. Foo [EGA94] and Modalys [EIC95] use a Scheme interpreter. Elody uses an interface to construct lambda expressions graphically [OFL97]. PatchWork offers a visual language on top of Common Lisp to construct functional expressions [AR93]. Its successor OpenMusic extends the graphical paradigm to include the object-oriented programming style of the Common Lisp Object System [Ass96]. Other systems that include functional composition include DMix [Opp96].

Lisp environments are attractive because of their dynamic, interactive usage. They offer high-level abstractions and allow the user to concentrate on the algorithms rather than on execution flow and data typing (without abandoning type checking, though). Music environments treat many more problems than just temporal composition. They generally incorparate a rich set of tools, libraries, and language constructs that formalize musical theories, most of which are in the realm of pitch organization. This topic is out of the scope of this work.

Dannenberg uses this functional approach to describe composite structures that may vary according to the context. He uses the terms "parametrized behaviors" or "behavioral abstractions." The problem used as an example is that of the drum-roll. Consider we described a composite structure that contains the drum line of a piece and repeats fifty notes. What happens if it is incorporated into a score and then stretched by a factor of two? Are the notes stretched by a factor of two, or are there another fifty notes added to the end? When the composer describes the drum line in a parametrized or algorithmic manner, both solutions are possible [Dan89]. A similar problem is that of the grace note. A grace is played quickly before another note and serves as an ornament. The grace note depends on the following note and could be grouped with it in a composite object. However, when we stretch this combined element, the grace note should keep the same duration, since it is played just as quickly in any context.

The possibility to specify the contents of a composite object in terms of its position and duration within the piece is a desirable feature. It is also found in the "maquette" editor proposed by Assayag & Agon. In this editor containers, temporal objects, and patches (graphically defined composition algorithms) are placed in time. The start time and duration of an object are available explicitly. An object can use these time parameters to construct their musical contents, but can also assign new values to them. In addition, links can be drawn graphically between the objects in the editor. These links are used to pass data among the objects during the evaluation and scheduling of the structure. Logical time (as opposed to execution time) becomes an explicit parameter in the construction of the piece [Ass96].

The use of functional or algorithmic descriptions of pieces does not seem as wide spread in multimedia authoring tools as it is in music. This may be due to the difference in number and type of objects manipulated. The number of components used in the editing of multimedia documents is in general less than the number of notes found in a music composition. Furthermore, every component in video editing, for example, has an associated semantics to it, something that is less prevalent in music where the basic elements are fairly abstract.

Most music environments calculate the structure of a piece statically. The stucture is created from a combination of functional descriptions and initial data. Once created, the internal relations that lead to the structure are lost. What remains is a data-only representation. When a more complete description is available the internal relations may be verified after each editing. Consider the user conceives a piece and describes its structure using functional methods. The user then request a visual representation of the piece. If the user edits the visual representation, the internal relations that characterize the structure may no longer be verified. Whenever he evaluates the description a second time, the modifications are lost. S/he can mofidy the algorithms that generate the structure but this may not be trivial. The parallel and sequential organizations provide another illustration of these constraints. If a composite structure is defined as a sequence this property should be verified after all future transformations.

Little work has been done on this subject in the field of music composition. However, propositions on this topic can be found in the field of multimedia composition. The solutions include the use of constraint networks to check the temporal properties of a composition [FJLV98, JLR$^+$98].

When the composition lays the basic scheme of an interactive piece, not all the start times and durations of the temporal objects are known in advance. User actions may trigger events that in their turn signal the start or the end of other temporal objects. A static description of the structure of the piece no longer satisfies the requirements of dynamic, interactive presentations. Duda & Keramane propose the use of causal relations between objects instead of descriptive relations such as those proposed by Allen. They attach *actions* to the boundaries of an interval [DK94, KD97]. New intervals are constructed using interval expressions such as "a seq b," which construct a new interval in which a and b form a sequence. So far nothing new. However, since actions are attached to the beginning and the end of an interval, the above expression is interpreted as: "when a stops, it causes b to start." The relation between a and b is causal and no longer descriptive. The advantage of this approach is that when the duration of a is unknown or modified, the relations between a and b still hold. Thus, the relation is resistent to local time transformations.

Thus far, we have talked about the temporal organization of discrete elements. However, the representation of music structures also requires values that vary "continuously" in time. We will discuss some of the issues in the representation and manipulation of continuous time functions in the next section.

3.2 Continuous time functions

A full discription of a music piece requires the use of both events and continuously varying parameters. However, music programs using only discrete events exist. Since most elements in common music practice, such as pitch and intensity, are discretized, a discrete description can suffice for a composition environment. The MIDI protocol uses events to send control values to synthesizers. These control values may describe typical "continuous" parameters such as air pressure. The rationale behind this presentation is the these parameters are slow varying and can thus be sampled at a low rate. Despite this fact, this "pointillist" representation makes any transformation (stretching, transposition, adding vibrato, ...) on the pitch curve difficult. Systems using only continuous functions have also been proposed [MMR74]. But when continuous functions are used to describe the start and end time of sounds, the duration of the sound is hard to express.

The need for both discrete elements and continuous functions is all the more desirable in a music environment that integrates composition and sound synthesis. Sound synthesis systems are expected to offer a rich set of continuous functions to describe the evolution of the control parameters in time. Continuous time functions describe frequency and amplitude curves, and any other variable in the synthesis algorithm that may vary in time. In this section we will consider the following issues:

- The relations between continuous time and events.
- The relation between continuous control functions and hierarchical structures.
- The manipulation of continuous time functions.

There is a close dependency between event times and continuous time functions. Event times influence the definition of continuous functions. For example, an amplitude envelope should be stretched to fit in the duration of a note. This type of relation is not always valid, however. Amplitude curves of percussive instruments are independent of the note duration. Continuous functions can also specify event times. The user may wish to express the end of a note in terms of its amplitude. As in the case where the note should stop when the amplitude drops below -60 decibels. Also tempo curves influence event times. Tempo curves are

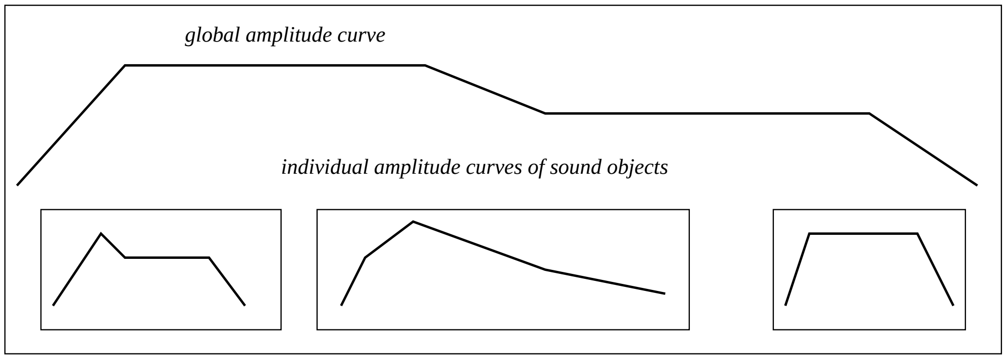

Figure 3.3: *The local amplitude curves of the temporal objects are shaped by a global amplitude curve.*

continuous functions used to introduce tempo changes and rhythmic alterations and phrasing. Continuous functions are also used to define more local stretch operations and time deformation [AK89, Dan97].

There is often a correspondance between control functions and the hierarchical structure of a piece. Control data is passed between the different levels within the piece. Classical examples are the phrasing of a group of notes. Phrasing may apply intensity and pitch changes global to all notes of the group. Several temporal objects can have a global amplitude curve shaping their local amplitudes (Fig. 3.3). In the case of the global amplitude curve, data is passed “top-down”. There are cases in which several objects are engaged in a transformation and data is passed between objects. One of those is a *portamento* between two sound objects (Fig. 3.4). Pitch information of the two sound object has to be known to some higher level transformation function. In addition, the control function must anticipate the value of the second note.

Rodet, Cointe, and collegues developed an environment to control the Chant synthesizer for the singing voice. They introduced the notion of synthesis-by-rule to control the transitions between vowels and notes. In Formes a piece is represented as a tree structure of objects representing time-varying values. Rules associated with the objects calculate the output values at time intervals defined by the system. These rules are invoked by a monitor object that walks this “calculation tree” [RPB93, RC84]. The hierarchical structure and succesive invocation of the rules of “parent” and “child” objects gives an elegant solution to phrasing problems. The more recent Diphone project inherits the hierarchical

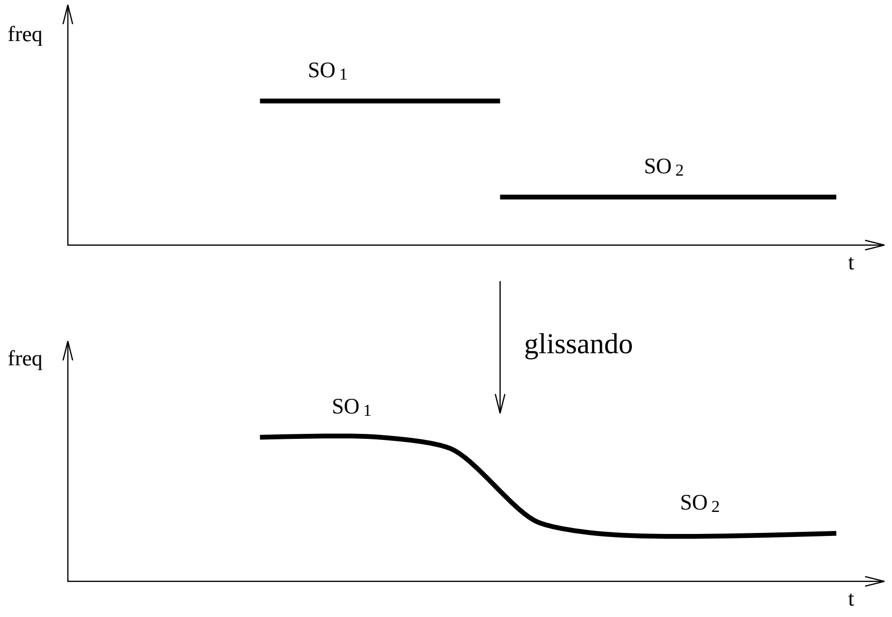

Figure 3.4: *A glissando between two temporal objects requires the acces to data local to each temporal object by a global transformation function*

organization for the description and interpolation of phrases from Formes but no longer offers the lisp interface [RL97].

Anderson & Kuivila also allow the hierarchical structuring of control functions. Control values are calculated in time by "processes." Each process has its local virtual time space. This allows local time deformations. The values are calculated at well-defined times, often at the beginning or end of a note. They are not used for fine-grain control. Time functions are calculated incrementally: they expect the next time to be bigger than the current time. This complicates the anticipation of control values [AK89].

Also Foo, developed by Eckel & González-Arroyo, defines a rich set of constructs to define continuous control functions. In addition, time functions can be multi-dimensional. Hierarchical time contexts can be constructed. A context represents a time offset to its parent context and defines a temporal closure for all the synthesis modules in the context.

The manipulation of continuous functions for music systems brings forth its own set of problems. We will present examples taken from a series of articles written by Dannenberg, Desain, and Honing [DH92, Hon93, Hon95, Dan97, DDH97]. Consider a sound with a vibrato (Fig. 3.5 a). Vibrato is generally considered as

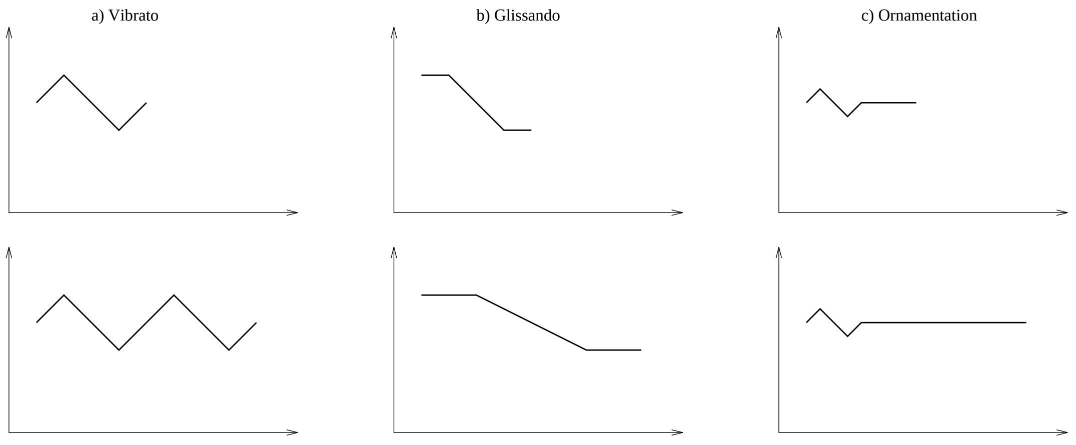

Figure 3.5: *The vibrato problem*

the regular modulation of the frequency around the perceived pitch of a note. The vibrato is characterized by the frequency and the amplitude of the modulation. The frequency of the vibrato is independent of the duration of a sound object and thus invariable under time transformations such as stretching. When the sound object is stretched, more vibrato cycles are added at the end. A (sinusoidal) glissando, however, depends on the duration of the sound object. In a glissando the frequency of a note "slides." Glissando should be stretched accordingly when the temporal object's duration changes (Fig. 3.5 b). An ornamentation such as the one depicted in figure 3.5 c) has a constant duration. It is not stretched, and no cycles are added at the end. The representation and handling of these different behaviours is known in the literature as the vibrato problem. They can be considered as equivalents of the drum-roll and grace note problem, discussed earlier, in the organization of discrete elements. In the vibrato problem, the behavior of the stretch transformation is local to the temporal object. In the case of the global amplitude curve or the portamento, structures on a more abstract level are engaged in the transformation.

Dannenberg developed a series of composition systems. Chronologically, these are Artic, Canon, Fugue, and Nyquist [DR86, Dan89, Dan93]. Artic and Fugue do not handle sound synthesis but provide a rich framework to define continuous values for the control of sound synthesis. In Fugue and its successor Nyquist, sound synthesis can be handled. In these environments basic musical elements can be combined into composite structures. The musical elements have a body and a transformation environment. For example, the body of a *note* structure contains its pitch and duration. The transformation environment contains transformation data such as stretch values. Time functions and transformations can refer to this environment parameters (see also [Hon95]).

Honing & Desain have defined a framework both for the composition of discrete musical elements and continous control functions. Both elements can form alternating layers of discrete and continuous information [DH92, Hon93]. They propose the use of "generalized time functions." These functions are defined as functions of three arguments: the actual time, a start time, and a duration. Generalized time functions can be combined, or passed as argument to other time functions. They can be linked to a specific musical attribute such as pitch or amplitude. Since the time context of a control function is available, solutions to the vibrato problem can be expressed elegantly.

Design and implementation

Chapter 4

The fundamentals of the architecture

The computer has extended musical writing both on the level of musical structure and on the level of the sound control. Complex musical ideas and structures can be formulated with the computer and its results tested and manipulated with an unpreceeded ease. Sounds can be produced that correspond exactly to a pre-described shape, evolution, or spectral contents. However, the bulk of the music applications still consider musical structure and sound synthesis as two distinct fields within the realm of musical composition. Most composition environments are focussed on "l'écriture" (musical writing). To give the composer an idea of the composition a score can be played using a MIDI instrument. However, if the composer wants to integrate synthesized sounds into the composition, s/he falls back on other applications for the synthesis. This has several drawbacks. Firstly, if the user can integrate a description of the sound synthesis in the composition, s/he cannot play them. Being able to play synthesized sounds from within the composition environment is desirable when the composer wants to integrate timbre in the compositional process. In that case, data structures controlling the sound synthesis are defined within the composition environment. In particular, real-time playback is very helpful for the composer. In Gareth Loy's words:

> It is stifling to compose without being able to hear near-term results. [. . .] In the case of sample-by-sample calculation of an acoustical waveform, all issues of how music sounds – its musical interpretation – must be resolved by the composer in addition to all strictly

> compositional issues. The effect of not having real-time feedback is like trying to play an instrument by touch only, and having to complete the performance before being able to hear a single note ... No possibility to reinject a human touch into non-real-time synthesis ... [Loy89], page 331.

Secondly, in the transfer to the synthesis application the structure of the composition is lost: it is reduced to an event list on a one-dimensional time line. This limits the possibilities of interactive pieces, since the context information is no longer available. Lastly, the user has to copy the data to the synthesis application. In the case of spectral data this may represent a significant overhead and a large amount of memory space.

This still existing barrier between musical writing and sound sculpting is artificial. Both are concerned with the description of structure and data generation. There should not be fixed sounds on the one hand, and structural organization on the other. The sounds should be formed as a function of their place in a certain context. In addition, the description of the sound synthesis may affect higher levels of the organization. Integrating the two fields is necessary if composers want to benefit fully from the possibilities offered by computers to create multimedia pieces.

Another remark is that a lot of sound synthesis systems use a primitive set of data types and only pass around constant values. Most composition environments, however, are build on top of high-level language interpreters and handle functions as "first order citizens." Maybe because of this fact, interactivity in real-time systems is often reduced to setting the value of a controller. It is not possible to re-program the synthesizer during runtime. The distinction between editing and execution is very clear in real-time applications. The usage of a synthesis systems proceeds in two steps. In the first step the composition or synthesis patch is loaded. In the second step the synthesis patch is executed. No editing or inspection during runtime is possible.

If we ask the question how come composition environments and synthesis systems take on such a different approach, we find only one reason. And it is a technical one. Most composition-oriented environments are written on top of a high-level language interpreter, generally a Lisp-style language. These environments delegate all storage reclamation issues to a garbage collector. Systems for sound synthesis, especially those offering real-time performance, are written in C or C++ for efficiency. They handle memory reclamation explicitly. Integrating both functionalities means combining garbage collection, real-time, and multi-threading in one system. In this thesis we propose and discuss the implementation of such an integrated environment. Part of this thesis will therefore focus on the real-time versus garbage collection issue. In addition, we will design the environment to make a seamless navigation between synthesis, composition, and interactivity possible. In the design of the integrated environment we must consider the following aspects carefully:

- The choice of a language both for the implementation and interaction with the user,
- The model that serves as a basis for the environment,
- The definition of the interfaces for basic components such as synthesis modules, control functions, and sound output,
- A rich model for the representation of time and the organization of a piece.

On the one hand, the complexity of describing synthesis patches, music structures, and the system's response to user input make the use of a high-level language necessary. Even if we provide convenient tools such as graphical editors, expert users request the possibility to extend and customize the environment. On the other hand, it must be possible to express signal processing techniques efficiently in the language we choose to develop in. It must be possible to write a hard real-time synthesizer. The choice of the language will determine both the flexibility and the real-time capabilities of the system.

The conceptual model of our architecture must assure that interactivity, sound synthesis, and composition can be combined in the same environment. For example, music improvisation requires the modification of multimedia material in runtime. We want the existing tools for composition to be available for the interactive system. We must therefore assure that the compositional structures are "interaction-aware." Similarly, the data structures must be general to include many possible synthesis techniques and compositional paradigms and, at the same time, capture enough information to be useful.

Instead of copying and converting complex control structures between several systems, we want the data to be fully exchangeable between all components of the environment. The same control structures that the composer manipulates in the score are used during the synthesis. This is important to allow the manipulation of timbre as a musical element in a composition.

Some synthesizers have no notion of time at all (except for the notion of *now*); others use a one-dimensional time line. To make compositional structures more dynamic for use in interactive pieces, a representation of the temporal information is needed during runtime.

In this chapter we discuss a new architecture for multimedia systems. The architecture is designed to integrate various paradigms of time-based media handling. In particular, it covers synthesis/rendering, composition, and live interaction. We consider the seamless integration of the three mentioned approaches as an important achievement of the presented work. Our rationale behind the necessity of this combination is that these different approaches depend upon each other. Obviously, to perform a piece, it has to be composed, but also, compositional structures guide interactive systems. User intervention can also cause a (re-)organization of the composition. To allow all possible schemes we need an integrated environment. The system we propose in this text combines real-time sound synthesis, an embedded Scheme interpreter, and interactive event handling. The guideline in the design of our system is a fairly simple model that we will expose in this chapter.

Several existing projects aim at the same goals as ours. Projects like Formes [RC84], Nyquist [Dan93], and Foo [EGA94] guide our work. PatchWork [AR93], OpenMusic [Ass96], Csound [Ver86], and Max/FTS [Puc91b, Puc91a] are milestones in the field of computer music and greatly inspire us. Our proposal is inevitably an integration of existing designs. However, it is made possible only by nowadays program techniques and computing power. There are several additional reasons why our model is different from existing music systems. First, as stated above, it closely integrates three elements rarely found together: composition, synthesis, and real-time. Second, our model is very dynamic. The user is no longer restricted to a two step process of programming the environment and executing the synthesis program. Instead, we offer a fully dynamic and interactive environment in which the user can create and modify data structures during the execution. Thirdly, in comparison to some synthesis programs, our model handles very rich data types, including functional objects. This is true for the arguments carried by the events as well as for the structures manipulated by the synthesis or composition algorithms. Lastly, the underlying model does not explicitly mention sound synthesis. It can be used for any time-dependent, rich media that needs to be output at regular time intervals.

The rest of the thesis is organized as follows. First, we discuss why we have chosen to develop the environment in the Java language and argument the use of an embedded Scheme interpreter (Section 4.1). Before we present the model formally (Section 4.3), we introduce some of the concepts used throughout the text (Section 4.2).

The discussion of the implementation is divided into two chapters. Chapter 5 describes the basic components of the architecture and the major classes for the sound synthesis, control, and output. Chapter 6 discusses the structures for the the organization of discrete elements and continuous functions in time. Finally, in chapter 7, we estimate the real-time behavior of the system and discuss the influence of the garbage collector.

4.1 The choice of Java and an embedded Scheme interpreter

We have chosen the Java language for the development of our environment. In addition, we embedded a Scheme interpreter into the environment. In this section we will argument this choice. First, let us underline the advantageous of an embedded interpreter (see also [BS95] for a discussion on the subject.)

The advantages of an embedded interpreter can be grouped in four sections:

- High-level language: Interpreters generally provide an accessible syntax that reduces the learning curve and facilitates the reading and coding of programs. The language is more concise and allows to express simple programs rapidly.

- Portability: The interpreter hides machine specific features. Interpreted languages are therefore machine independent.

- Development environment: Interpreters allow to extend the behavior of the application in ways that were not programmed by the developer. In addition, they promote a short programming cycle, in general a read-eval-print loop, in which the compilation phase is invisible. They provide a better error handling than in traditional languages, and handle memory management implicitly.

- Distribution: Due to the textual format, concise syntax, and dynamic compilation, source code and data structures can easily be sent over the network.

A distinction has to be made between an *extendible* and an *embeddable* interpreter. We will say an interpreter is *extendible* if it provides the means to extend its primitives. To this extend it provides a *foreign interface*. This is a programming interface that allows the following:

- call objects implemented in a language other than the language used by the interpreter,

- insert new primitives to the interpreter: define new functional objects in the interpreted language to access functions provided by an external library.

In our project the use of an interpreter that is extendible is important to integrate new synthesis modules into the environment.

An interpreter is *embeddable* if it is possible to evaluate an expression of the interpreter language from another language. Embeddable interpreters are generally extendible. When we use "embeddable interpreter" we assume the interpreter is extendible as well.

The use of an embeddable interpreter will prove to be important: external libraries can invoke the evaluation of complex interpreted functions. For example, the response of the system to low-level events can be programmed by the user in a high-level language and is by no means fixed by the environment.

However, there are a couple of drawbacks to the use of interpreters:

- Performance

- Complexity

- Different memory management

- Incompatible object systems.

The performances offered by interpreted languages are still below those offered by traditional languages.

In certain cases it can be a complex development task to embed an interpreter into an environment. In addition, the interpreter forces the user to pick up programming, something s/he may not be acquainted with. This can be help by supplementing the environment with graphical editors (as in the case of PatchWork and OpenMusic). However, such an interface constitutes a research project of its own [Ago98]. The expert user who wants to get the best of the system will have to invest the time to master the language.

The interpreter uses a garbage collected memory heap; external libraries written in traditional languages use explicit memory. To define new primitives in a foreign language, a new type is defined, accompanied with a set of functions to print and construct these objects. Additional functions have to be written for the coordination between the garbage collection and the native memory management. Imagine an external object refers to an object of the interpreter and that this reference is the only link to the object. Since there is no link from the interpreter's data structures to the object, the storage space of this object will be reclaimed incorrectly. The external libraries must either help the garbage collector during the tracing or explicitly tell the interpreter when objects are used and released. This places a burden on the developer.

In addition, the interpreter typically uses an object system that is distinct from the object system used by the external libraries. This fact, combined with the separate memory strategy, makes it impossible the share objects between the two subsystems seamlessly.

Let us illustrate these problems with a concrete example. We experimented for a while with the Extension Language Kit (Elk, [LC94]), which is also used for the Foo and Modalys environments. The first problem was that Elk is not multithreaded. This problem can be overcome, but is not trivial. Second, if we want Elk objects to be referred to by our code we should use a single memory strategy. We have therefore put Elk on top of Boehm's collector for C [Boe93]. This solved the memory management problem but still left us with two other problems. First, it is difficult to make Boehm's collector real-time. Read and write barriers are hard to implement for C code. In general virtual memory protection is used as read or write barrier, but this strategy cannot be made real-time. Second, we still have two separate object systems. This means that the root of Elk's type system has nothing to do with the root of the type system in the external library. This prevents a transparent use of objects, unless we recode the external library.

A really integrated, seamless solution would require the following steps:

1. Write a portable runtime layer,
2. Define the structures for the garbage collector,
3. Define a class hierarchy,
4. Write an interpreter in this framework,
5. Develop external libraries in the framework.

This project clearly requires a lot of work and is very difficult to realize.

A solution to the problem is available since the introduction of the Java platform by Sun Microsystems. The Java platform is a combination of a virtual machine, an object-oriented language, and rich set of standard libraries [GJS96, GM95]. In addition, a Scheme interpreter, called Kawa, is available. The Scheme language is a small but powerful, Lisp-style language [KCR98]. The Kawa implementation is entirely written in the Java language and provides a foreign function interface to call Java objects from within Scheme. Since all the Scheme primitives are implemented as Java objects, they can easily migrate to the external libraries. Kawa achieves good performances by compiling Scheme expressions to Java byte code which is then immediately executed on the Java virtual machine [Bot98].

The inconveniences of this solution are reduced efficiency compared to traditional languages. The performance of Java is below that of C. The efficiency is better with the use of just-in-time compilers (JIT) and is still improving. In its current state, Java does not offer real-time guarantees. However, Sun announced real-time extensions late 1998. Because of the high interest in Java for embedded systems a lot of effort is put on real-time Java.

The advantage of this solution is the single memory management strategy and a single object system. In addition, we benefit from Java's portability, standard

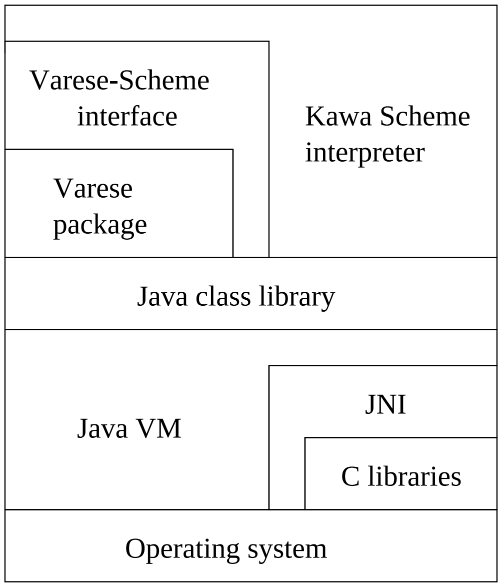

Figure 4.1: *The global view of the environment.*

libraries, and improved error handling over traditional languages. Libraries written in C, for example existing signal processing libraries, can still be called thru the Java Native Interface (JNI).

The global view of the environment is given in figure 4.1. The Java classes of our project are grouped in a package called Varèse, after the French-American composer. We have written an interface in Scheme to call our classes from within the Scheme interpreter and to use Scheme primitives in the Varèse environment. In the following sections we discuss some of the basic concepts that underlie our system's architecture.

4.2 Synthesis processes, events, and programs

Theoretically, with the use of computer, any sound can be produced. The digital representation of a sound, however, asks for an enormous amount of data and it is therefore inconceivable to generate this data by hand. To describe and facilitate the generation of sound synthesis algorithms are used. A pure sound, for example, can be generated as follows:

$$s(t) \quad = \quad a(t)\,\sin\left(\phi(t)\right)$$

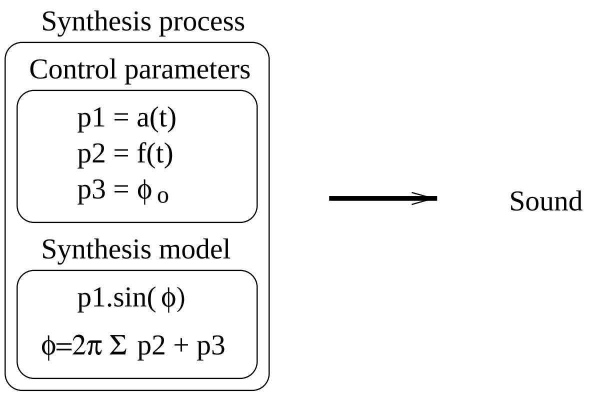

Figure 4.2: *To produce sound we need the combination of a synthesis model and control parameters. We will call the combination of the model and the parameters a* synthesis process.

$$\phi(t) \quad = \quad \phi_0 + 2\pi \int_0^t f(t)dt$$

We can distinguish two parts in the expression above: the *synthesis model*, and the *control parameters*. The synthesis model defines the behavior of the synthesis process under various inputs and determines the characteristics of the sounds that can be produced. The control parameters steer the model to generate one specific sound. For a pure sound we can isolate three control parameters: $p_1 = a(t)$, $p_2 = f(t)$, and $p_3 = \phi_0$. The amplitude envelop $a(t)$ and the frequency contour $f(t)$ are continuous functions in time. The initial phase ϕ_0 is a real number. The model is described by $s(t) = p_1 \sin(\phi(t)), \phi(t) = p_3 + 2\pi \int p_2 dt$. All sounds produced by the model will be pure sounds. The sound characteristics that we can "compose" are the pitch contour and the amplitude envelope. (The critical reader will immediately imagine complex control functions with which the output will not be a pure sound. In this introduction, for the sake of conciseness, we will accept slowly varying control functions.)

We will call the combination of a synthesis model and the control parameters a *synthesis process* (Fig. 4.2). The activation of a synthesis process will generate a sound signal. Calling a synthesis process with a time value t will return the value $sp(t)$ of the synthesis model at that time. For digital sound, synthesis processes are called at discrete times, multiples of the sampling period T_s: $sp(nT_s)$. For efficiency, synthesis processes are not called for every sample, but for an array of N samples. In the latter case, the array of samples is passed as argument: $sp(nNT_s, out)$.

The system would be quite incomplete if we allowed only one synthesis process to be active at one time; the system therefore accepts several simultaneous synthesis processes. Furthermore, a synthesis process necessarily starts and ends at some point in time. We will say the synthesis process is *alive* or *active* during a certain laps of time. We do not want to impose any restrictions on the start and end time. The live span of a synthesis processes can be very short, or can take the full length of the composition. The start and end time, together with the parameters of the synthesis process, can be known before the execution. This is the case when the system performs a composed piece. In an interactive set-up, however, these elements can be determined or modified in runtime by user intervention. Discrete user input is called an *event* and provokes some action on the side of the system. The action that needs to be taken is described by a *program*. The side effects of the evaluation of a program can be trivial, such as stopping a synthesis process. However, programs can also perform complex tasks, such as a complete reconfiguration of the system or the generation of a part of the composition. In this text we make extensive use of the concept of program. It is loosely defined as a series of instructions to perform a task, given a number of arguments. We accept, for example, that programs implements synthesis models or create new data structures. In the next section we will formalize these ideas.

4.3 A formal description of the architecture

In this section we will give a formal description of a multimedia system. This formal description is not meant as a rigid mathematical theory. In this section we will express the concepts of the environment in a concrete notation. This model is intended as a guideline for the implementation. We can briefly describe the system in words as follows.

1. The system produces a series of output values at regular time intervals. In the case of a sound system, the output is a sample frame or a buffer of sample frames. The output value is calculated by a set of *synthesis processes*. Each synthesis process is a *program* and its output depends on the state of the system.

2. At any time *events* may arrive. These events change the state of the system according to some program. By changing the state of the system, the events influence the output values calculated by the synthesis processes.

We will precise these ideas below. Let us start with a more precise definition of events. Let $\mathbf{P_n}$ be the set of n-ary functional symbols, called programs, $\mathbf{P_n} = \{p_i\}, n, i \in \mathbf{N}, n \geq 0$. We will call $\mathbf{P} = \bigcup \mathbf{P_n}$ the set of programs. Furthermore, let $\mathbf{V}$ be the set of anything.

Definition 1 (Event) *An event e is a triplet $\langle t, p, \bar{a} \rangle$, where $t \in \mathbf{R}_+$ indicates the time of the event, $p \in \mathbf{P_n}$ is a program, and $\bar{a}$ is a vector of values $(a_1, a_2, \ldots, a_m), a_i \in \mathbf{V}, m = n$. Let $\mathbf{E}$ be the set of all events.*

Using the previous definitions, we can now define the synthesis system as follows.

Definition 2 (Synthesis system) *The system S is a 6-tuplet $\langle \tau, p_i, o, SP, E, V \rangle$. Where:*

$\tau \in \mathbf{R}_+$ *indicates the time interval between outputs,*
$p_i \in \mathbf{P_0}$ *is the initialization program,*
$o \in \mathbf{V}$ *is the output of the system,*
$SP \subset \mathbf{P_2}$ *is the set of active synthesis processes* $SP = \{sp_i | sp_i \in \mathbf{P_2}\}$,
$E \subset \mathbf{E}$ *is the set of events,*
$V \subset \mathbf{V}$ *is a set of anything, called the environment.*

We will call the state of the system $S = \langle \tau, p_i, o, SP, E, V \rangle$ the quadruple $\langle o, SP, E, V \rangle$. The state of the system can change thru the application of a program p with arguments $\bar{a}$. In particular, the program p can create and insert new events in the system. To simplify the notation in the following discussion we will notate the set of events created by the program p as E_p. We notate the application of a program p as follows:

$$\langle o, SP, E, V \rangle \quad \overset{p(\bar{a})}{\rightarrow} \quad \langle o', SP', E \cup E_p, V' \rangle$$

With p_{out} we will denote the current composition of the elements of SP: $p_{out} = (sp_n \circ sp_{n-1} \circ \ldots \circ sp_1)$. We will call p_{out} the system's *output program.* An event e whose program is the output program p_{out} is called a synthesis event, $e = \langle t, p_{out}, \bar{a} \rangle$.

In the previous paragraph we gave the definition of the elements of our system. We will now see how the system evolves. We assume that external to the system there exists a clock that indicates the precise time t. The system then follows three rules:

Rule 1 (Initialization)

$$\frac{t = 0}{\langle 0, \emptyset, \emptyset, \emptyset \rangle \quad \overset{p_i()}{\rightarrow} \quad \langle o_0, SP, \{\langle k\tau, p_{out}, (k\tau, o)\rangle, k \geq 0\} \cup E_{p_i}, V \rangle}$$

Rule 1 describes the initialization. It is applied when the time equals zero. First, the set of events E is made equal to the set of all synthesis events. These synthesis events schedule the evaluation of the synthesis program p_{out} after every interval τ. The arguments to p_{out} are the time $k\tau$ and the output o of the system. Then the initialization program p_i is evaluated. p_i defines the initial state of the system and can create additional events and synthesis processes. After initialization the system evolves according to rules 2 and 3.

Rule 2 (Synthesis)

$$\frac{t = k\tau \ \wedge \ e = \langle k\tau, p_{out}, (k\tau, o)\rangle \in E}{\langle o_{k-1}, SP, E, V \rangle \quad \overset{p_{out}(k\tau, o)}{\rightarrow} \quad \langle o_k, SP, E \setminus \{e\} \cup E_{p_{out}}, V \rangle}$$

Rule 2 describes the synthesis task. The synthesis is triggered by the synthesis events $\langle k\tau, p_{out}, (k\tau, o)\rangle$. When the time equals $k\tau$ the output program is evaluated. The main purpose of the program p_{out} is to calculate a new value for the output o. However, it can also modify the state of the system and insert new events in E.

Rule 3 (Event handling)

$$\frac{t = t_e \ \wedge \ e = \langle t_e, p, \bar{a} \rangle \in E \ \wedge \ \langle t_e, p_{out}, (o) \rangle \notin E}{\langle o_k, SP, E, V \rangle \quad \overset{p(\bar{a})}{\rightarrow} \quad \langle o_k, SP', E \setminus \{e\} \cup E_p, V' \rangle}$$

Rule 3 describes the event handling. If at time $t = t_e$ there exists an event $e = \langle t_e, p, a \rangle \in E$, the program p of the event is evaluated with the arguments $\bar{a}$. The program p can change the state of the system and schedule new events. If an events exists at the same time of a synthesis event, the synthesis event is handled first.

The output o of the system is updated by the output program p_{out} at times $t = k\tau$. Let o_k be the value of o after the evaluation of p_{out} at $t = k\tau$.

Definition 3 (Output series) *The output series O of the system is the series of successive output values o_k, or $O : o_0, o_1, \ldots, o_n, \ldots$*

For both sound and animation it is advantageous to think of the output of the system as the combination of the output of several, simultaneous sources, objects, or activities. Each activity represents one easily definable element in the total output. Moreover, each activity can start and stop at any time. Thus, at any time any number of simultaneous activities can exist. A synthesis process describes the action performed by one such single activity. It is modeled as a program $sp \in \mathbf{P_2}$. The application of the synthesis process modifies the value of the output o given as argument. The final value of the output of the system is a combination of the effect of all active synthesis processes. The program p_{out} describes the combined effect of all the synthesis processes. The order in which the synthesis processes are evaluated depends on the domain. For a graphical system the order depends on the visual hierarchy of the graphical objects and/or layers. For a sound synthesis system the output values are mixed (added) together.

During the execution of a piece, synthesis processes can be inserted or removed from the set of active synthesis processes. This insertion or removal is the result of the evaluation of an event. For convenience, we will introduce the programs *add* and *kill*. They are defined as follows: $add(sp) : SP \leftarrow SP \cup \{sp\}$ and $kill(sp) : SP \leftarrow SP \backslash \{sp\}$. Similarly, we will call an event $e = \langle t, p, a \rangle \in \mathbf{E}$ an *add-event* (*kill-event*) if $p \equiv add$ ($p \equiv kill$).

We will implement the model described above in a framework combined of the Java environment enriched with an embedded Scheme interpreter. Since the synthesis processes perform all the signal processing code, they should be as optimized as possible. We will develop them in the Java language because it provides better performance than Scheme code. Events have less stringent time requirements than the synthesis processes. The programs carried by the events can be written in Scheme. How the system is changed in response to events can be defined thru the Scheme interpreter. Although synthesis processes are written in Java, they can, however, request the evaluation of Scheme procedures by sending events. This is the case for schedulers, which provoke the evaluation of events program at their given times.

The Scheme interpreter typically uses a table to store the defined procedures and variables. This table is typically called the *environment*. Looking back at the model described above, we see that this environment corresponds to the environment V of the system. Thru the Scheme shell the user defines and manipulates variables defined in the environment. Since, the synthesis processes depend upon those variables, the user can interact with the synthesis processes thru the shell.

In our model, we deliberately neglected one element that is crucial for the implementation: the evaluation of a program *takes time*. In this model the evaluation of a program p is assumed to be instantaneous. Since our multimedia system is time bound and the hardware on which it runs has limited resources (limited CPU power), we will have to consider the duration of the evaluation. There exist scheduling algebras that take into account the duration of operations (see for example [vGR98]). The complexity of the model would increase unnecessarily if the notion of duration was included. Instead we will consider this duration in the practical situation of the implementation. We will see how both the synthesis and the event handling will be modeled as concurrent tasks. In chapter 7 we will discuss the time behavior of these tasks.

Besides an interface we have to provide the user with additional language constructs and data structures to organize a music piece. Just programs and events is too austere to express the complex structure of a multimedia piece. We need tools to organize data and specify the structural relationships that exist between the elements of the piece. These data structures will be discussed in the chapters 5 and 6.

Chapter 5

The implementation and basic classes of the environment

In the previous chapter we proposed a formal model of the system. Three types of objects form the basis of our system: synthesis processes, events, and programs. So far the environment is very Spartan. If we want to offer the user a convivial workspace for multimedia composition, we will need to include additional data structures and functionality. This and the following chapters address this issue. We will also give examples of how the user interacts with the system.

5.1 The pillars of the architecture

In this section we discuss the main components of the system. We will build the system using an object-oriented approach. In particular, the current prototype is completely written in the Java programming language. Considering the formal model, and, adding to that the Scheme shell, we can isolate three distinct tasks: the synthesis, the event handling, and the interaction with the user. The three tasks execute simultaneously; we will embed each of them in its own thread. One object, called Varèse, sets up all the tasks. The first task is the sound synthesis. The system handles this in real-time, which means that the synthesis and the output are interleaved. One buffer of sample frames is synthesized then written to the output device. The *synthesizer* holds an array of references to active synthesis processes (Fig. 5.1). A new synthesis process can be added to the array by sending an *add* message to the synthesis thread. A *kill* message

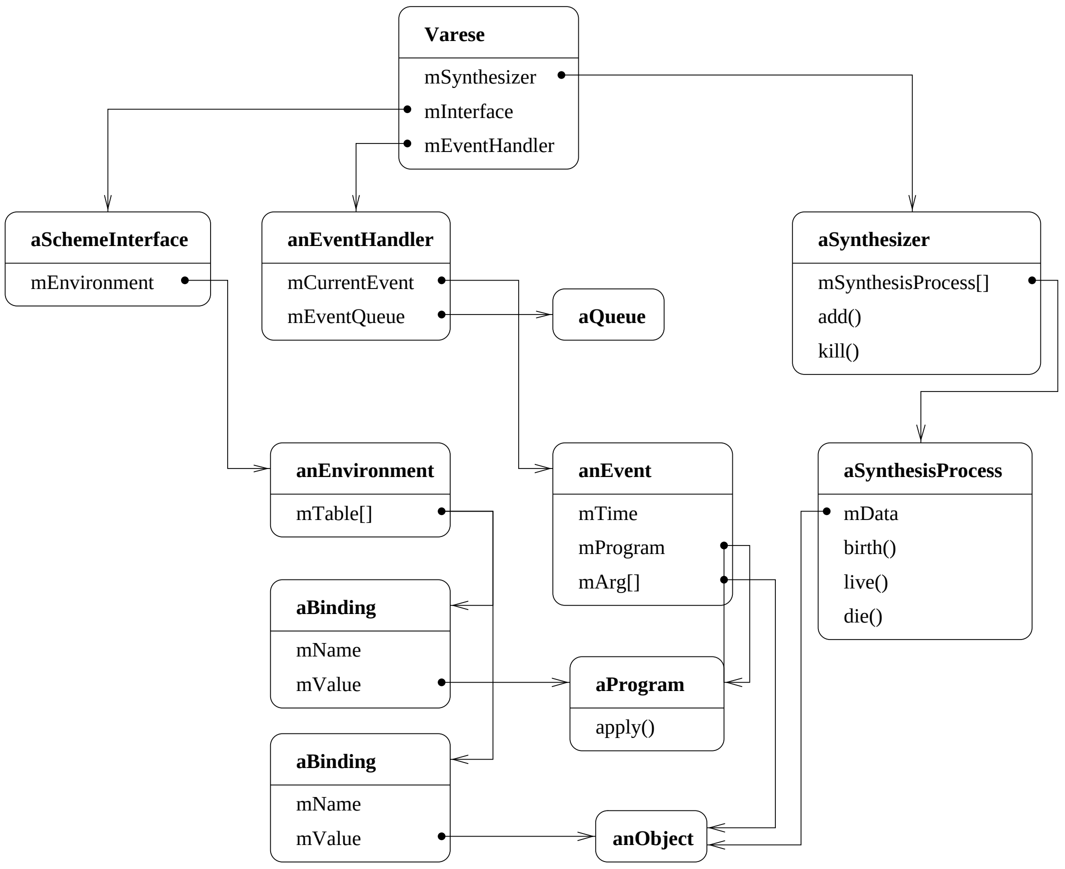

Figure 5.1: *Object diagram of the multimedia system: The three main tasks of the system are the synthesis, the event handling, and the user interaction (Scheme interpreter). Each task is embedded in its own thread.*

will make an end to the scheduling of a synthesis process and remove it from the array. Each synthesis process produces sound according to its own algorithm. The synthesizer calls the active synthesis processes every τ seconds and passes it a reference to the output buffer. The synthesis process can manipulate the buffer according to its algorithm. The synthesizer then writes the buffer to the output device. A particular synthesis process is a subclass of the abstract class `SynthesisProcess`. The three key methods of this class are: `birth`, `live`, and `die`. Subclasses must implement these three methods that define the synthesis algorithm the process embodies. We will discuss these methods in more detail in chapter 7.

The second task the system handles is the event processing. At any time, asynchronous to the sound synthesis, events can arrive. Events can be sent from schedulers, programs, or synthesis processes. Events are processed by the *event handler*. New events are posted to an *events queue* (figure 5.1). The event handler picks the first event out of the queue, removes it from the queue, and applies the event's program with the given arguments. The program can create, add, and kill synthesis processes. It can modify the state of the system on which the synthesis processes depend, and thus influence the output of the synthesis. It can also define new events, variables, or programs. The time indication of the event is meant mainly for event schedulers. It is ignored by the event handler: once the event is posted to the event queue, it will be handled as soon as possible. The event's program must implement the method `apply` defined by the interface `Program`.

The last task is the interaction with the user. The music system we present in this text incorporates a Scheme interpreter. This interpreter allows the dynamic definition of both variables and programs. Composers interact with the interpreter to create musical structures, algorithms, synthesis processes, and control structures. A discussion and example of some these structures is given in this and the following chapter.

The text entered by the user in the Scheme shell is parsed and evaluated by the interpreter. The evaluation of textual expression can be seen as event handling, and this is why the formal model in the previous chapter does not explicitly mention the user interaction. To call the Java objects from within the shell we have defined a set of Scheme functions. This interface will be discussed in section 5.2.

We have three basic tasks (synthesis, event handling, and the interpreter) and three basic objects (programs, events, and synthesis processes) in our environment. The event handler processes the events by applying their program. The interpreter parses and evaluates "textual" events. The synthesizer handles the synthesis by calling upon all the active synthesis processes. So far, we have not detailed these synthesis processes very much. In section 5.3 we discuss how to describe and control complex synthesis processes. In section 5.4 we discuss how this environment can be used in distributed applications. In this case all external interaction with the environment – event handling, Scheme interaction, and sound output – is handled over the network.

5.2 The Scheme interface

The interface between the Scheme interpreter and the Varèse package consists of the following:

- a set of class encapsulating Scheme primitives,
- Scheme functions to create and call objects of the Varèse package.

Among the classes that encapsulate Scheme primitives we note the class `SchemeProgram`. This class implements the `Program` interface and refers to the Scheme primitive *procedure*. Functions defined in the Scheme environment can thus be used in the event handling.

We have defined Scheme procedures to interact with the synthesizer thru the Scheme shell. Since all sound synthesis is done by synthesis processes the first procedures we need are those to add or kill a synthesis process. The user can add a new synthesis process to the synthesizer using the following syntax:

```
(add <id> <sp>)
```

The `id` is an integer number, specified by the user, used to identify a synthesis process. The identification number is specified by the sender of an events and not by the synthesizer object to avoid the overhead of asynchronous communication.

The second argument, `sp`, is an object of type `SynthesisProcess`. To kill the process the following expression is used:

```
(kill <id>)
```

Creating and scheduling a new event is done as follows:

```
(event <t> <p> <a1> ... )
```

The procedure `p` will be evaluated at time `t` with arguments `a1`, ... The procedure `p` is automatically wrapped in a `SchemeProgram` object.

In the following section we will describe the major classes of our project. We will show examples of their use and introduce the Scheme functions to interface them. We do not want to overload the text with the definition of all the Scheme interface functions we will use. The reader can find a reference of all the Scheme functions in appendix A.

5.3 The description and control of sound synthesis

Multimedia often depends on a continuous change in time. This is certainly true for sound synthesis where amplitude envelopes describe the value of the amplitude in time. We start our discription of the classes for sound synthesis with a discussion of control functions.

5.3.1 The control functions

Most synthesizers define *controllers*. In many systems the controllers push their values down to the synthesis processes. The rate at which they output their values is fixed and defined by the system. Our definition of control functions is more general and resembles that found in Foo. These are the characteristics to which we attach a lot of importance:

- Continuous: control functions are considered continuous in time. They can return a value for any given time.

- No predefined control rate: Control functions do not send there values down to the synthesis modules, but instead their values can be requested by any other object.
- Reentrant: The value of a controller can be requested in any time order and not necessarily with increasing time values.
- Multi-dimensional: Control functions always return an array of values. This allows complex control functions (for example, partials of additive synthesis) to be manipulated as a single object.

We defined an interface, called `ControlFunction` comprised of two methods: `value`, and `dimension`. The `value` method returns the value of the control function. Its argument is a time in seconds. The `dimension` method returns the dimension of the control function.

The constant control function outputs a constant array of values. The name "constant" refers to the fact that its value is time-independent, not to the fact that the user cannot change its value. With the following functions constant controllers can be created and their values modified:

```
; Create a new constant controller of dimension 2
; and values [440,880].
(define c (const 440 880))

; Set the zero'th value to 261.
(const-set! c 261 0)
```

The breakpoint function is a piece-wise linear function. The following expression creates a new breakpoint function of dimension 1. At time 0 it has a value of 0, at time 0.5 value 1, and at time 3 value 0.

```
(bpf 1 '((0 0) (0.5 1) (3 0))))
```

The sine-wave controller outputs a sinusoidally varying value. It requires a control function describing its frequency and a control function describing its amplitude:

```
; A sine-wave controller with a frequency of 6Hz
; and an amplitude of 0.1
(ctrl-sin (const 6) (const 0.1))
```

Control functions can be combined to create more complex control structures. `ctrl-add` and `ctrl-mul` respectively add and multiply two control functions. The function in the following example constructs a controller modulating a frequency value sinusoidally. With this controller a vibrato effect can be obtained.

```
(define (vibrato freq mod-freq mod-amp)
  (ctrl-mul (const freq)
            (ctrl-add (const 1.0) (ctrl-sin (const mod-freq)
                                             (const mod-amp)))))
```

5.3.2 The description of synthesis algorithms

The type `SynthesisProcess` is an abstract class. As discussed in section 5.1, all defined synthesis processes are subclasses of this class. Consider we define a class `Sinewave` that inherits from `SynthesisProcess` and implements the sine-wave oscillator described in section 4.2. If we neglect the initial phase of the oscillator, we can create a new instance of the oscillator and pass it two arguments for its creation: the frequency curve and the amplitude envelope. The Scheme procedure to realize this looks like this:

```
(sinewave <freq> <amp>)
```

The `freq` and `amp` arguments are continuous functions in time describing the evolution of the frequency and amplitude, respectively. Combining the previous definitions, we can add (and kill) a sine-wave oscillator playing at a constant frequency typing:

```
(add 0 (sinewave (const 440) (const 0.1)))
(kill 0)
```

The evaluation of `add` and `kill` is done immediately after the user enters the text. With the help of two additional procedures (`add-sinewave` and `kill-sinewave`) we can schedule the add and kill of the sinewave as follows:

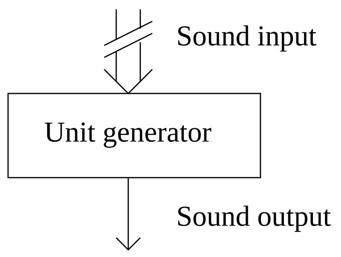

Figure 5.2: *An unit generator taking as input a number of sound signal and returning a sound signal as output.*

```
(define (add-sinewave id freq amp)
  (add id (sinewave (const freq) (const amp))))

(define (kill-sinewave id)
  (kill id))

(event 1.0 add-sinewave 0 440 0.1)
(event 2.0 kill-sinewave 0)
```

We will need some way to construct more complex synthesis processes. The sine-wave oscillator used above will not please the musical ear for very long. Not only should developers be able to create new synthesis processes that embody a particular synthesis technique. Also the expert user should have the possibility to construct new synthesis techniques from within the Scheme shell. A priori, the formal model in the previous chapter gives no clue how to create synthesis processes. That is exactly what we expect from our model. Indeed, we specify only the general interface of a synthesis process. It is up to developers of applications, interfaces, or signal processing techniques to realize the synthesis techniques they need. It is not the aim of our project to write extensive digital signal processing algorithms. However, we have realized a set of basic synthesis objects and a way of defining new synthesis algorithms. We inspired ourself on Mathew's unit generators [Mat69]. This paradigm is still the most used to build synthesis patches. We would like to stress, however, that our system is not restricted to the unit generator approach. Indeed, using the flexibility of the Scheme language, we think any approach can be realized in our environment. We refer, for example, to Modalys [EIC95] that also uses a Scheme interface to define physical models.

We define the abstract class `UnitGenerator`. It provides a basic implementation of the interface defined by `SynthesisProcess`. In addition, a unit generator can take a number of sound inputs. A filter, for example, takes one sound signal as input and returns the filtered signal. The input can be any other object implementing the `SynthesisProcess` interface. Below we give a simple example of a sampler unit generator that is sent thru a bandpass filter:

```
; We define three control functions for the filter:
; the central frequency, the bandwidth, and the gain
; of the filter.
(define fc (const 200))
(define bw (const 50))
(define gain (const 1.0))

; We create a new bandpass filter with the control functions
; defined above.
(define filter (bandpass-filter fc bw gain))

; We load the sound file ``piano.aiff'' into a sample table.
; We indicate the base frequency of the stored samples.
(define piano-snd (sound-table ``piano.aiff'' 440))

; We create a sampler synthesis process. This sampler plays
; thru the sound table. The playback frequency and amplitude
; are defined by two control functions.
(define piano (sampler (const 440) (const 1.0) piano-snd))

; Connect the piano to the 0-th input of the filter.
(connect filter 0 piano)

; Pass the filter to the synthesizer to hear the filtered piano.
(add 0 filter)
```

The user can create synthesis processes dynamically by defining a new program. This program can be included in events. For example:

```
(define (add-piano id freq amp)
  (add id (connect (bandpass-filter fc bw gain) 0
                   (sampler (const freq) (const amp) piano-snd))))

(event 1.0 add-piano 0 440 0.1)
(event 2.0 kill 0)
```

In the examples above we explicitly defined the events, event times, event arguments, and synthesis process identification numbers. A composer does not always wish to reason in such concrete terms. It will be useful to introduce abstractions to work on a higher level of music organization. Furthermore, when we define these abstractions we will not only have to take into account the start and stop times of the resulting synthesis process, but also their control functions and other arguments. This fairly complicated problem of time organization will be handled in the next chapter.

5.3.3 Useful abstractions: Synthesis techniques and voices

Consider again the example we gave in the previous section:

```
(define fc (const 200))
(define bw (const 50))
(define gain (const 1.0))
(define piano-snd (sound-table ``piano.aiff'' 440))

(define (add-piano id freq amp)
  (add id (connect (bandpass-filter fc bw gain) 0
                   (sampler (const freq) (const amp) piano-snd))))
```

This example displays some shortcomings of the current strategy. First, the control functions `fc`, `bw`, and `gain` are defined globally and control all instances of `add-piano`. Imagine the composer wants to write a piece for two piano's. Either both piano's are controlled simultaneously by the control functions, or all control functions have to be copied for each piano. The former case will surely not be always the desired behavior. In the latter case, we pollute the name space of our environment with redundant controller names.

The second problem occurs when we want to create libraries of synthesis techniques. Imagine we have defined a synthesis technique called "filtered piano". The composer has no description, and maybe no access, to the parameters of the synthesis technique.

To solve these problems we adapt a meta-class like strategy. We introduce two structures, `SynthesisTechnique` and `SynthesisVoice`. Synthesis techniques create new synthesis voices; synthesis voices create new synthesis processes. They provide abstraction, encapsulation, and auto-documentation for the creation of synthesis processes. Synthesis techniques define the default parameter values for synthesis voices. Similarly, synthesis voices define the default parameter values for synthesis processes. Both classes allow to inspect and override the default values. In addition, all parameters have a name, and the user can ask a textual description for each of them. To create new synthesis processes, a synthesis voice requests a number of arguments. The number and type of arguments depends on the type of the synthesis voice, but we will accept that the first argument defines the frequency and the second argument defines the amplitude of the synthesis process. (This will be discussed in more detail in section 6.6.)

Users can define new synthesis techniques dynamically in the Scheme shell, or create libraries of synthesis techniques. For example:

```
(define vibra_a4
  (sound-table ‘‘vibra_a4_mono.aiff’’ 440.0))

; We define a procedure that creates new synthesis processes.
(define (make-vibra-process f a  env)
  (sampler vibra_a4 f (ctrl-mul a env)))

; We define a new synthesis technique, called ‘‘vibraphone’’.
; The synthesis technique defines three controllers ‘‘freq’’,
; ‘‘amp’’, and ‘‘envelope’’. The synthesis processes will be
; made with the make-vibra-process procedure defined above.
(define vibra-syntech
  (synthesis-technique
    "vibraphone" "vibraphone"
    make-vibra-process 3
```

Index		Number		Type		Data			
high	low	high	low	high	low	b0	b1	. . .	b n

Figure 5.3: *The structure of a UDP event: the first two bytes is the program index, the following two bytes counts the number of arguments, the following two bytes describe the type of arguments, the remaining bytes are the values of the arguments.*

```
    '(("freq"     "frequency (Hz)"                (const 440.0))
      ("amp"      "amplitude (linear)"            (const 0.1))
      ("envelope" "amplitude envelope (factor)" (const 1.0)))))

; We make a new voice from the vibra synthesis technique.
(define vibra-voice (synthesis-voice vibra-syntech))

; We override the default value of the envelope controller
; defined by the vibra synthesis technique.
(voice-set! vibra-voice "envelope" (bpf 1 '((0 0) (0.5 1) (3 0))))

; We make and add a new synthesis process with the vibra
; synthesis voice. We pass the frequency and amplitude as
; arguments.
(add 0 (voice-make vibra-voice (const 261.0) (const 0.5)))
```

5.4 Using the system in a distributed environment

To allow interfaces to be distributed, low-level events can be sent over the network using the datagram protocol (UDP). We have chosen a very straightforward binary format for the events for reasons of efficiency: the first two bytes indicate a program index number, the following two bytes count the number of arguments, the following two bytes code the type of the arguments (int, double, char, ...), and the rest are the arguments themselves (Fig. 5.3.) To receive UDP events, a new event thread is created that listens on a given port. The UDP event thread holds an array of programs set by the user. Whenever an event arrives the program index number and the argument count are extracted. Using the type indication the arguments are converted into objects. An array of characters is converted into a string. The program with the corresponding index is applied to handle the event. Using the power of the Scheme language, the user can program complex responses of the system to user events. The following example shows how to set-up and initialize the UDP event thread. The UDP thread waits for events that start and stop a note. The parameters of the note are MIDI-style pitch and velocity values.

```
; Create and start a UDP event thread, listening on port 9000
(define udp (udp-thread 9000))

; Define two procedures to start and kill a note
(define (noteon pitch vel)
  (add (sinewave pitch (const (midi->hz pitch))
                       (const (db->amp (/ (- vel 130) 2))))))
(define (noteoff pitch) (kill pitch))

; Pass the programs to the UDP thread.
(udp-thread-set! udp noteon 0)
(udp-thread-set! udp noteoff 1)
```

It is fairly trivial to connect the in-port and out-port of the Scheme shell to a TCP/IP socket. Network-aware applications can create a constant connection to the environment and provide a more interactive exchange with the environment than is possible with low-level events.

Also the sound can be sent over the network. The sound output uses a coder/decoder (codec) to convert the samples (an array of double values) to the desired output format (16 bits linear, 8 bits linear, μ-law, etc ...). The sound output then writes the samples into a data stream (see figure 5.4). For example, sound can be written to the audio device, to a file, or to a TCP/IP stream. Tests with sound output over a TCP/IP connection on a local network are satisfying. However, TCP/IP does not provide any control over the quality of service. We therefore envision to realize a sound output on top of the Real-time Transport Protocol (RTP) and Real-time Streaming Protocol (RTSP). In the future version

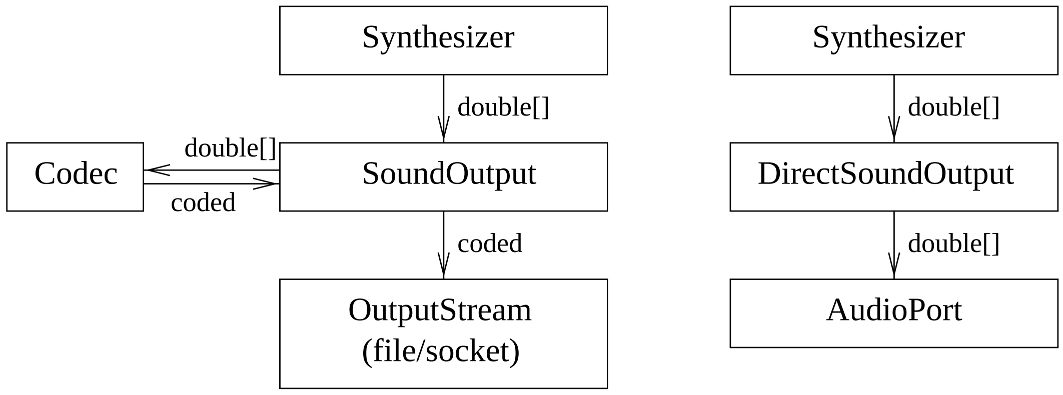

Figure 5.4: *The SoundOutput object uses a coder/decoder (Codec) to convert the buffer of samples in double format to the format used in the output. The converted samples are written into a data stream. For sound output directly to the sound device of the local machine, the DirectSoundOutput writes the double samples immediately the audio port that will convert the samples natively.*

we will also move the system on top of the Java Media Framework (JMF) and the JavaSound library, which have been released recently by Sun Microsystems. We will benefit from the synchronization mechanisms between several media players, the platform independance, the coders/decoders, and network capabilities (based on RTP/RTSP) offered by these frameworks.

A more complex interaction with the system requiring the passing of binary data can be build on top of Java's Remote Method Invocation (RMI) or on top of the Common Object Request Broker Architecture (CORBA, [Obj96, SV95, SGH$^+$97, Vin97]). An easy solution would be to define a new variable in the Scheme environment, passing its name and value. Both the integration with JMF and RMI are left for future work.

Another future project is to use several synthesis systems concurrently. The systems can be placed in series, in parallel, or a combination of both. Low-cost hardware can be combined to solve computation intensive performances. For example, the first computer system generates the sound, passes it on to the next system for a compresses before it is sent to the users/listeners on the network (Fig. 5.6).

In case the system is running on a multi-processor system, the synthesis task could be distributed over the available processors. The synthesizer then runs on

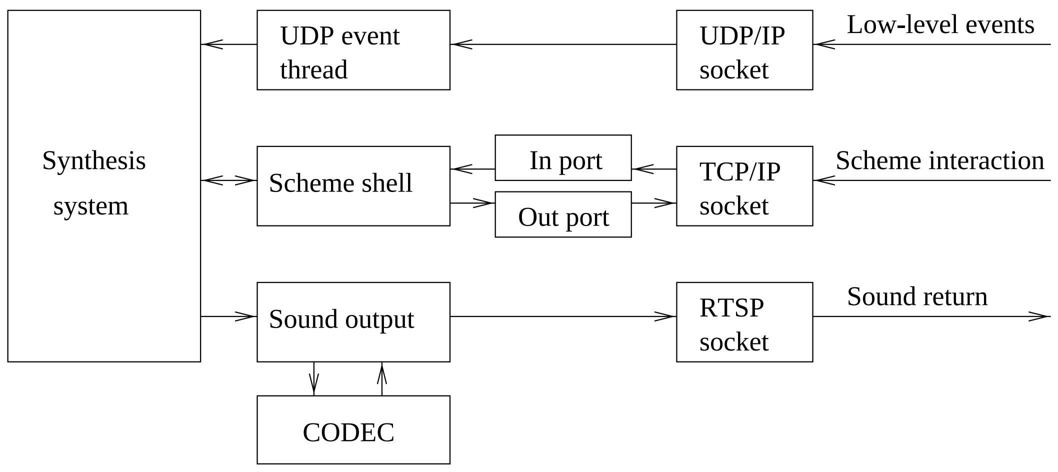

Figure 5.5: *The diagram of the system in a distributed environment.*

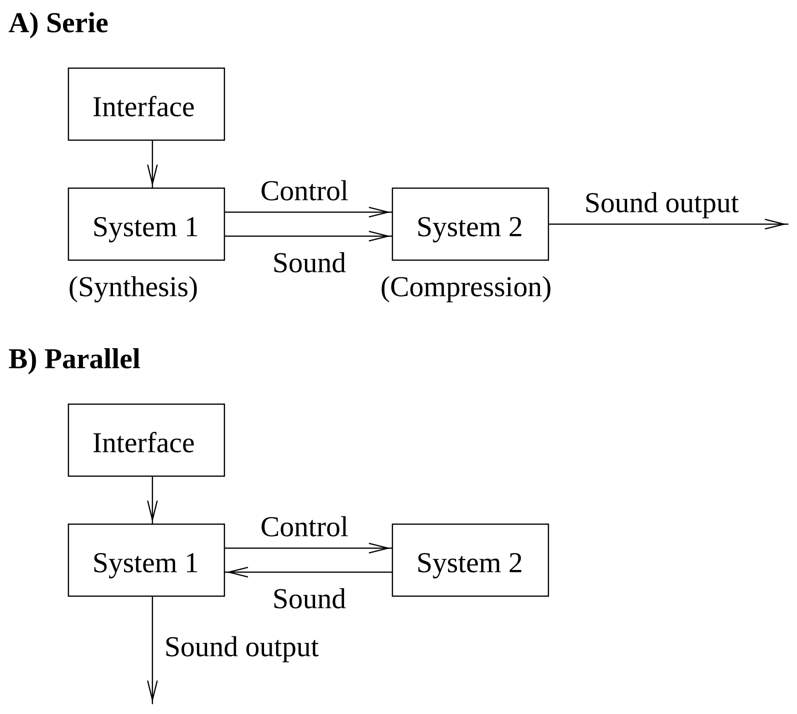

Figure 5.6: *Two systems used simultaneously can be placed in serie or in parallel.*

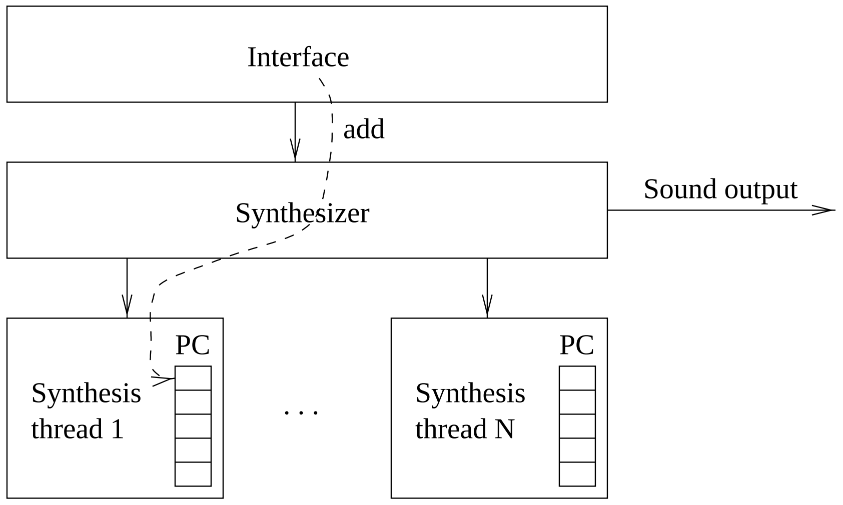

Figure 5.7: *The synthesis system, when run on a multiple processor computer can use several threads to divide the synthesis task over the available processors.*

top of several threads and distributes the synthesis processes over the existing threads to balance the load (Fig. 5.7.)

Chapter 6

Structures for composition and the organization of time

In the previous chapter we introduced structures to describe events, synthesis techniques, and control functions. The system still lacks structures to organize these elements into a coherent composition. Furthermore, the time, as it is represented in the formal model, is linear. However, in certain cases we want to relate the time in the composition to the "absolute time" in a non-trivial fashion. In this chapter we propose a set of classes to describe the relations and behavior in the time organization of a music piece. Our propositions are based on the works discussed in chapter 3. Many of the problems cited in that chapter find an elegant solution by adopting a rich time model. The first part of this chapter will therefore only consider time. This discourse will be applicable to any time-based media. In the second half of the chapter we will focus on the particular case of sound synthesis. We start with a description of activities.

6.1 Activities: The basic building blocks of a composition

When working with time-based media, the composer organizes "contents" in time. This contents can be any time dependent data: sound, video, animation, etc ... The contents must be played at a particular time and lasts for a certain laps of time, its duration. While composing, the composer pre-programs the

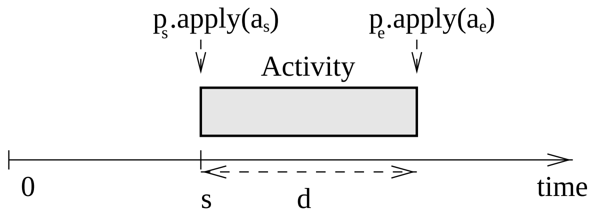

Figure 6.1: *An activity is triggered by a program p_s at a time position s, and ended by a program p_e, d seconds later.*

system: s/he defines the time dependent contents and describes their start positions and durations. With the term *activity* we will indicate the description and positioning of a such a time-based contents. We will say that at some point in time the activity is triggered at which point its contents is played. This activity has a certain life span after which it ends. Two major events mark the life of an activity: the first one starts the activity, the second one ends it. The definition of an activity is as follows (we recall that $\mathbf{V}$ is defined as the set of anything, see section 4.2):

Definition 4 (Activity) *An activity is a sextuplet $\langle s, d, p_s, \bar{a_s}, p_e, \bar{a_e} \rangle$. $s \in \mathbf{V}$ expresses the start position of the activity; $d \in \mathbf{R}_+$ is the duration of the activity. $p_s \in \mathbf{P_n}$ is a program applied with arguments $\bar{a_s}$ at the start of the activity. $p_e \in \mathbf{P_m}$ is a program applied with arguments $\bar{a_e}$ at the end of the activity.*

The program p_s of the activity, together with the arguments $\bar{a_s}$, describes the steps that need to be taken to start the activity. The program knows how to retrieve the contents of the activity and how to start it playing. Similarly, the program p_e and the arguments $\bar{a_e}$ describe how the activity is to be stopped (Fig. 6.1). The element s describes the start position of the activity. Note that the start position s is not necessarily expressed in a time unit for reasons that will become clear later in this chapter. Let us assume that an activity with a start position s has a start time s' expressed in seconds. To schedule an activity $\langle s, d, p_s, \bar{a_s}, p_e, \bar{a_e} \rangle$ two events are created and inserted in the event list of system: $e_s = \langle s', p_s, \bar{a_s} \rangle$ and $e_e = \langle s' + d, p_e, \bar{a_e} \rangle$.

The `Activity` class realized the structure described above. It is the root of most compositional elements we will discuss in this chapter. One of its principle methods is `play`, which will cause the evaluation of the start program and the scheduling of the stop program. The program p_s may need access to the exact start position and duration of the activity. This information is grouped in a structure called *time context* and will be discussed in detail further on in this text. The activity transparently inserts the time context as the first argument of the start program.

Consider for example, the activity $\langle s, d, add, (sp), kill, (sp)\rangle$. This activity, at its start time, adds the synthesis process sp to the synthesizer and kills it d seconds later. The Scheme interface defines all necessary interfaces to create activities, and set and get the fields of an activity. In our Scheme environment the activity above can be created as follows:

```
(define (start-sinewave context id freq amp)
  (add id (sinewave (const freq) (const amp))))

(define (stop-sinewave id)
  (kill id))

; Create a new activity starting at 0 seconds, with a duration
; of 2 seconds, that will start and stop a sinewave
(define sin-activity
  (activity 0 2 start-sinewave '(0 440 0.1)
              stop-sinewave '(0) ))

(play sin-activity)
```

In this example the start program is `start-sinewave`. The activity `sin-activity` calls this program with the additional parameters `(0 440 0.1)`. When we look at the definition of `start-sinewave`, we see that the context variable has been inserted implicitly. The programs p_s and p_e of the example have a simple side effect: the addition and removal of a synthesis process. The actions taken by the programs p_s and p_e can be far more sophisticated, however. For example, they can provoke the generation and scheduling of a part of the composition. The following code is only an illustration of how more complex composition algorithms can be included. We will assume the use of a composition algorithm `compose-something`.

```
(define (compose-something context)
  ; ... insert the composition algorithm here.
)

(define (stop-composition)
  ; ... do what ever is needed to clean up.
)

(define piece
  (activity 0 60 compose-something () stop-composition ()))

(play piece)
```

A composition comprises generally a great number of activities. It is the difficult task of the composer to specify the times, programs, and arguments of the activities. However, what intrigues both composers and music researchers most is the complex network of relationships that exists between the elements of a composition. Any of the elements of an activity $\langle s, d, p_s, \bar{a_s}, p_e, \bar{a_e} \rangle$ can be in complex relation to any set of elements of the other activities in the composition. The great challenge for the designer of composition environments is to provide the composer with a rich set of tools and language constructs that allow her/him to express and manipulate these multitude of relationships elegantly. In this chapter we will mainly concentrate on the time relations between the activities. In section 6.6 we consider activities that represent sound synthesis. In this case we can assume additional information on the types of arguments of the activities. We will start with the description of composite structures that maintain a set of relationships.

6.2 The organization of a piece with patterns and motifs

In this section we concentrate on the relationships between the positions and the durations of activities. Imagine the composer has created a piece. S/he then wants to apply a transformation to a section of the piece. A stretch, for example. Instead of completely recalculating the structure (which may not always be possible if parts of the structure are created manually) it is advantageous to propagate the modifications incrementally. This was already expressed by Honing in his article [Hon93]:

> Musical structure should be associated with time intervals. Constraints on these time intervals model the specific musical construct 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.

A similar problem is encountered in the construction of graphical interfaces. Basic graphical components (buttons, text fields, icons, ...) are grouped into containers. The layout of a container depends on the locations and sizes of the components. When the container is resized (because the window is resized, for example) new locations and sizes of the components must be calculated. The graphical containers in Java's Abstract Window Toolkit (AWT) delegate the task of positioning and resizing the components to a layout manager associated to the container. Furthermore, to add a graphical component to the container a high level description of the location of the component is given instead of the precise coordinates. The insert operation is handed over to the layout manager who interprets the description and places the component at the correct location with a correct size. The model has a double advantage:

Delegation: The behavior of the container under various manipulations is described separately. Instead of designing a container for every possible behavior, one general container now suffices. The container combined with the appropriate layout manager results in the wanted behavior. Providing the most reoccuring layout types will satisfy most of the users. More demanding users can create a new layout type without changing the architecture of the framework.

Abstraction: The location of the components can be specified with an abstract description, provided it is understood by the layout manager. Java's standard BorderLayout allows a component to be inserted in the "center", "south", "north", "east", or "west" of the container.

It is clear that a similar behavior is useful for time structures. We will inspire ourselves on these concepts for the organization of activities in time. Lately, we have learned that the user interface research community has since long used techniques based on local constraint propagation to specify layouts and the relations between graphical objects. Since our project was in advanced state, we leave a detailed study of these techniques for future work (the following articles can serve as a entry: [BMSX97, FB93]). More recently, these techniques have also found there entry into the realm of temporal composition of multimedia documents (see for example [FJLV98]). In the following text, we do not pretend to use techniques of logical constraint programming. However, it is in this direction that we envision our structures to evolve.

We will introduce composite structures that maintain a set of relations. The description of the relationships consists of two parts. First, we need to group all the activities that are related. Second, we need to describe the behavior of this group of activities when modifications are made. After a modification the system must verify that the relationships between the activities are still satisfied. The activities may need to be rearranged (change their start position and durations). In addition, we introduce two notions to cope with the particularities of music composition. We note:

Redundancy: It is frequent in music to repeat the same sequence in a piece. These sequences are not independent one of another but refer to a single, composed musical element.

Time deformation: Techniques of time warping or time mapping may deform local time axes. When activities are organized in time, these deformation may have to be taken into account.

The basic element of organization is called called a *pattern*. The pattern holds a reference to all the participating activities and describes the relationships between them. When a modification is made to the group or to an activity, the pattern will re-organize the activities such that their internal relationships are satisfied again. A pattern has to be notified of the following modifications:

- the addition of a new activity,
- the removal of an activity,

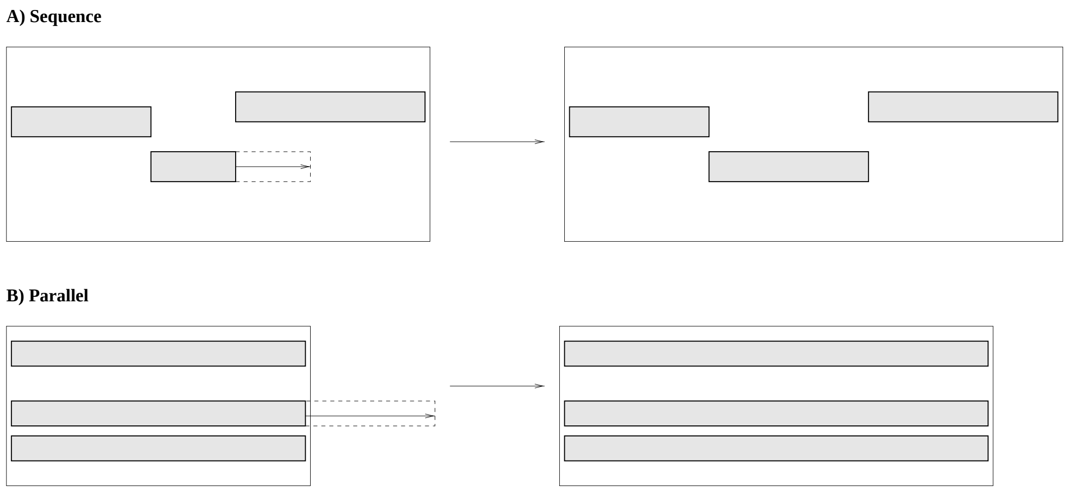

Figure 6.2: *Two examples of re-organization after a change of duration for A) a sequence pattern, B) a parallel pattern.*

- the change of the start position of an activity,
- the change of the duration of an activity.

Consider the parallel and sequence organization. The sequence places the activities consecutively; the parallel organization places them simultaneously. When one of the activities changes its duration the behavior of both organizations is different. This is depicted in figure 6.2.

When we add an activity it is placed into a pattern using its start position and its duration. The definition of activities does not specify how the start position is expressed. Indeed, any type of description may do, as long as the pattern understands it. Let us give some examples. The sequence organization adds a newly inserted activity at the end of the last activity in the pattern. Similarly, the parallel organization places the activity at the start of the pattern. For both organizations, no description of the start position is needed: the position is implicit. In many music systems the start position is expressed as the exact time in seconds. In that case the description is a real value expressing the start time of the activities. However, activities can also be added with the description "second activity." Or, "the position equals twice the duration." It really does not matter what the description is, as long as the pattern can figure it out and put the activity in the right place. At some point the start locations of the activities must be converted into a precise time expressed in seconds. The pattern has this

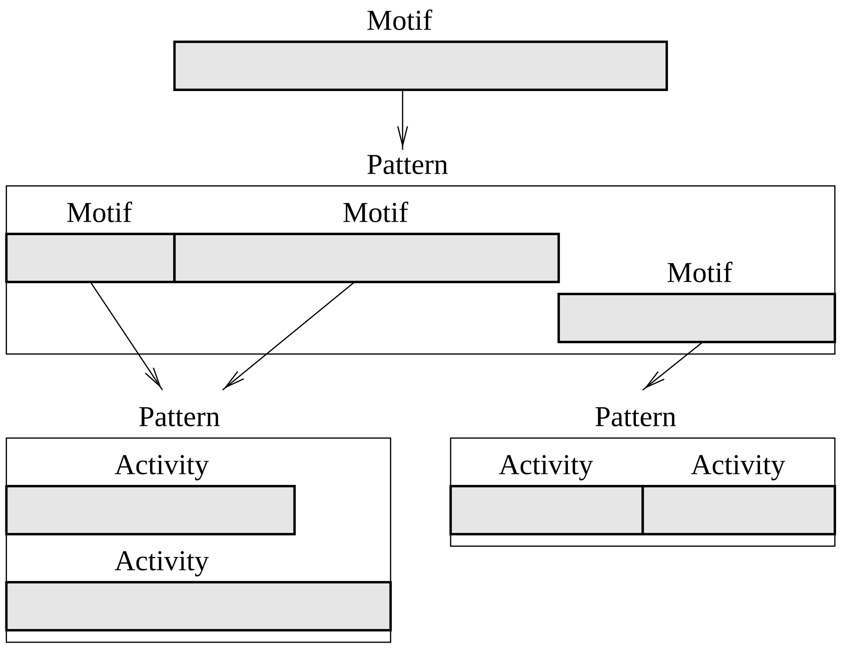

Figure 6.3: *An example of patterns and motifs*

responsibility, since it is the only instance that knows how to deduce the exact time position. How and when this conversion takes place is discussed next.

Patterns are not directly included into a music piece. There is a good reason for this. It is common in music to repeat a pattern throughout the piece. Each occurrence of the pattern might be submitted to slight variations. However, when modifications are made to the original pattern, these should be reflected by all occurrences. Buxton and colleagues already used the idea of *instantiation* to cope with the redundancy in music pieces [BRP$^+$78]. We introduce the notion of *motif*. Motifs are placeholders for patterns. They are a subclass of activities and have a start location and a duration. Their contents is defined by a pattern. We will say that a motif *subscribes* to a particular pattern. Whenever the pattern changes, the motif is notified and can update its contents. Since a motif is also an activity, it can be included in other patterns. The described organization groups activities into patterns; motifs represent patterns as activities and can themselves be included in patterns. These structures promote an hierarchical organization of the multimedia piece in a directed graph (Fig. 6.3). Patterns, motifs, and activities interact when changes are made and when the piece is scheduled to be played. The proposed organization assures that the internal relations defining

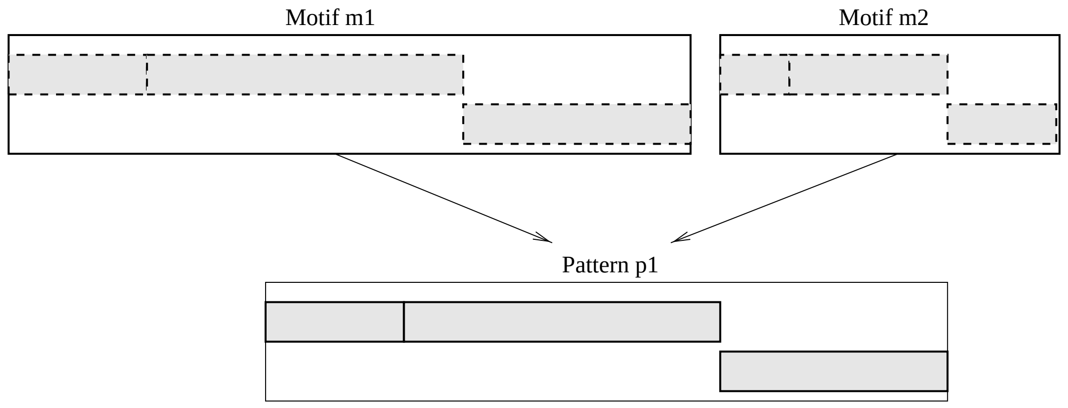

Figure 6.4: *Several motifs can subscribe to the same pattern. The pattern "clones" its contained activities for each motif and organizes the activities according to the duration of the motifs.*

the composition are satisfied after transformations. In addition, adjustments to a structure are incremental. Motifs can also be lazy: they only request the contents of the pattern when they really need it, for example when the piece is scheduled for playback.

Figure 6.5 shows the interactions between the elements when the start position or duration of an activity is changed. The pattern will rearrange the activities in the group as to satisfy all the relationships. This re-organization may trickle down and affect other patterns.

Figure 6.6 shows how the hierarchical structure of patterns and motifs is scheduled. This operation is needed when the user want to perform the piece. Since the duration of motifs are variable, the motif must request the pattern to organize its elements for the motif's duration (see Fig. 6.4). The motif "clones" the contained activities for the motif and gives them the appropriate start position and duration.

A piece can be scheduled dynamically or statically. When it is scheduled dynamically a motif schedules its contained activities when its start program is called. This allows late modifications to the pattern to be incorporated into the result. The disadvantage of this strategy is the application of a CPU and memory consuming scheduling task in runtime. When the structure is scheduled statically all events are created before the piece is played. This does not allow for last minute changes but is more time efficient at runtime.

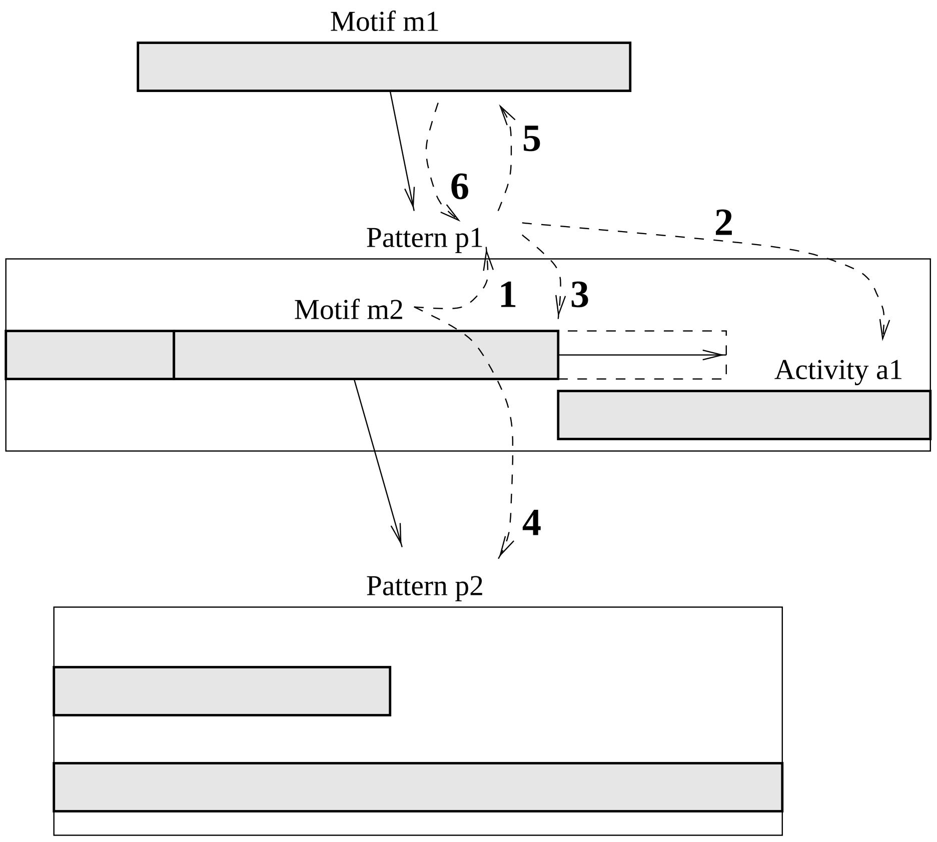

Figure 6.5: *The figure displays an example of the interactions between motifs and patterns. 1) When motif* $m2$ *is stretched it requests its container pattern* $p1$ *to change its duration. This request may fail if the internal relations of* $p1$ *can not be satisfied with the new duration. 2) and 3) Pattern* $p1$ *re-organizes its contained activities: it moves activity* $a1$ *and confirms* $m2$ *of its new duration. 4) Motif* $m2$ *requests its pattern* $p2$ *to update* $m2$*'s contents for its new duration. 5) Pattern* $p1$ *notifies motif* $m1$ *that its contents has changed. 6) Motif* $m1$ *may react to this notification by immediately requesting* $p1$ *the new internal organization.*

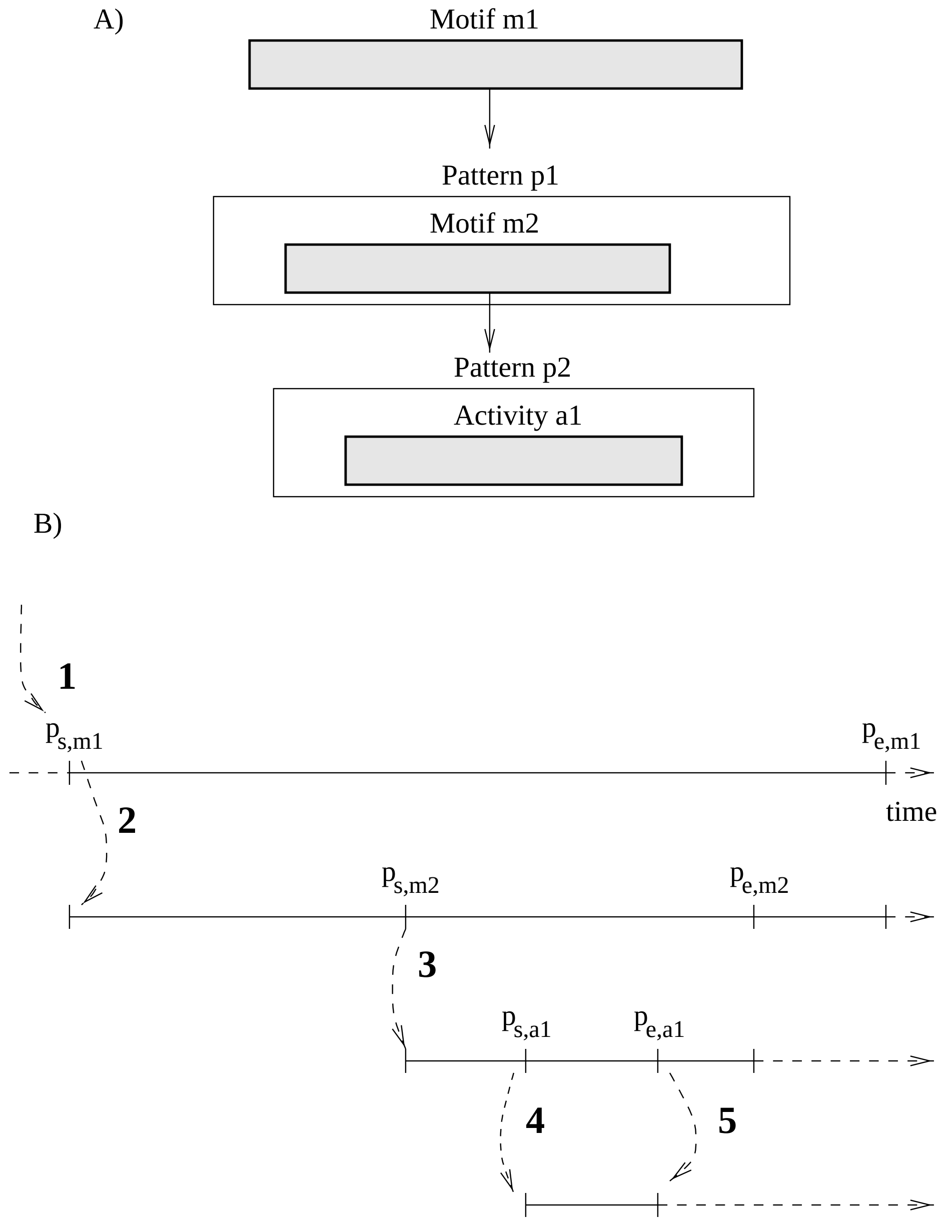

Figure 6.6: *Figure A) displays a simple organization of motifs, patterns, and activities. Figure B) shows the scheduling of events and the actions taken in runtime. 1) Motif* $m1$ *schedules two events:* $p_{s,m1}$ *and* $p_{e,m1}$*. 2)* $m1$*'s start program* $p_{s,m1}$ *calls pattern* $p1$ *to schedule its activities. Pattern* $p1$ *creates the start and end events of motif* $m2$*. This scheduling is done in runtime. 3) The application of the program* $p_{s,m2}$ *provokes the scheduling of activity* $a1$*. 4)* $a1$*'s start program* $p_{s,a1}$ *creates a new synthesis process and adds it to the synthesizer. 5)* $a1$*'s stop program* $p_{e,a1}$ *kills the synthesis process.*

The following set of Scheme procedures are some of the basic interface functions to create motifs and manipulate patterns.

```
(motif <start> <duration> <pattern>)
(get-pattern <motif>)
(pattern-add <pattern> <activity>)
```

The following functions creates a sequence pattern:

```
(sequence-pattern)
```

With these basic functions, combined with the functions to access the member variables of activities we can easily create a set of handy functions. The procedure `sequence` creates a motif and sets its pattern to the sequence pattern; `sequence-add` adds an activity to this "sequential motif"; `stretch` multiplies the duration of an activity with the given factor:

```
(define s (sequence 2))
(sequence-add (make-any-kind-of-activity 0 1))
(sequence-add (make-any-kind-of-activity 0 1))

(stretch s 2)
```

The above example adds two activities (with start at 0 and duration of 1) into the sequence. The start location of both activities is ignored by the sequence since it is defined implicitly by the pattern. The durations of the activities are scaled to the duration of the duration of the sequence (two seconds in this case). When the sequence is stretched, the contained activities are stretched accordingly.

Patterns define the organization of a set of activities; motifs reproduce a pattern in a particular context. In certain cases it can be interesting to describe "variations" to a pattern. We will detail this in the next section.

6.3 Defining modifications to patterns and motifs

The internal relationships between the elements of a pattern define its organization. The structures we propose do not fix these relationships for the user. In addition, time models provide a means to adjust the organization of a pattern thru time deformation. However, in some cases it may be necessary to include additional means of manipulation:

- to allow changes to other elements than the time dimension,
- to submit the activities in the pattern to some sort of processing before they are handed to the motif,
- to let the motif make a variation on the organization received from a pattern.

How the patterns and their activities are adjusted will likely depend on the contents they represent. In section 6.6 we will discuss the cases of activities and patterns that represent sound. In that particular case we will make some additional assumptions about the activities. However, we do want to provide a means to modify patterns in the general case. To solve the problem we defined the interfaces `Modifiable` and `Modifier`.

A pattern or motif that is modifiable implements the `Modifiable` interface. This interface defines three methods: `addModifier`, `removeModifier`, and `getModifiers`. Variations are delegated to another object called *modifiers*. A modifier implements the `Modifier` interface.

The interaction between the patterns, motifs, and modifiers is as follows. When a motif requests its pattern to obtain its contents, the pattern organize its activities according to the motif's duration. If the pattern is modifiable and has modifiers, it calls them one at a time. Each modifier can make adjustments to the organization, or to the contents of the activities as pleases. Then the activities are handed to the motif and the same scenario is repeated. If the motif is modifiable and has modifiers, it calls them in turn to adjust the activities. Finally, the activities are scheduled for playback. We will see concrete examples in the section on sound activities.

Although we have said that parts of the composition can be scheduled dynamically, it is not the most appropriate way to include interactivity into a piece. Changes to the composition during playback may lead to inconsistencies: what happens when we start playing a structure and then move its end time before its start time? Interactivity can be best integrated into the composition using causal relations, described next.

6.4 The use of causal relations between activities

When a composer writes a piece for an interactive installation the duration of the activities can depend on user actions. In that case, the duration is unknown at the time of the composition. Duda & Keramane propose the use of causal relations between activities [DK94, KD97].

The definition of causal relations in the proposed environment rests on two features:

- The start and end of an activity is defined by a program,
- We can define the sequential application of the start and end programs in the Scheme interpreter.

Below we give a short example of the sequential structure, this time based upon a causal relation. We define two activities `a` and `b`. The duration of `a` is set arbitrarily long. We use fictive procedures for the start end stop programs of `a` and `b`.

```
(define very-long 1000000)
(define a (activity 0 very-long start-a () stop-a ()))
(define b (activity 0 5 start-b () stop-b ()))
```

We define a help procedure, `fun-compose`, that realizes the sequential application of the procedures (with arguments) given as parameter:

```
(define (fun-compose p1 a1 p2 a2)
  (apply p1 a1)
  (apply p2 a2))
```

With the function above we can define the causal sequential structure as follows: (The constant `#!null` is defined by the Kawa implementation and is equal to the `null` value in Java.)

```
(define (causal-sequence a b)
  (let ((start-prog-a (get-start-prog a))
        (start-arg-a (get-start-arg a))
        (stop-prog-a (get-stop-prog a))
        (stop-arg-a (get-stop-arg a))
        (start (get-activity-start a))
        (dur (+ (get-activity-duration a)
                (get-activity-duration b))))
    (set-stop-prog! a fun-compose)
    (set-stop-arg! a stop-prog-a stop-arg-a play b)
    (activity start dur start-prog-a start-arg-a #!null ())))

(play (causal-sequence a b))
```

The function `causal-sequence` returns a new activity that can again be included into other structures. When the causal sequence is played, the end of activity `a` is determined by an event coming from an external source. The stop program of `a` causes both `a` to stop and `b` to start playing. The example shows the advantage of using a high-level interpreter. The expert user who wants to describe a more complex behavior of the system can pick up the code above and dynamically redefine the relations.

So far, we have discussed activities, patterns, and motifs. We dealt with discrete times only: the time locations and durations of activities. In the next section we will consider continuous time.

6.5 Introducing complex time dependencies in a composition

In certain cases we want to relate the logical time in the composition to the "absolute time" in a non-trivial fashion. Time deformations are useful in the following cases:

- It is easier to reason about the organization of a composite structure on a "linear" time axis followed by a projection of the organization with a non-linear "warping" function, then it is to reason directly on a "non-linear" time axis.

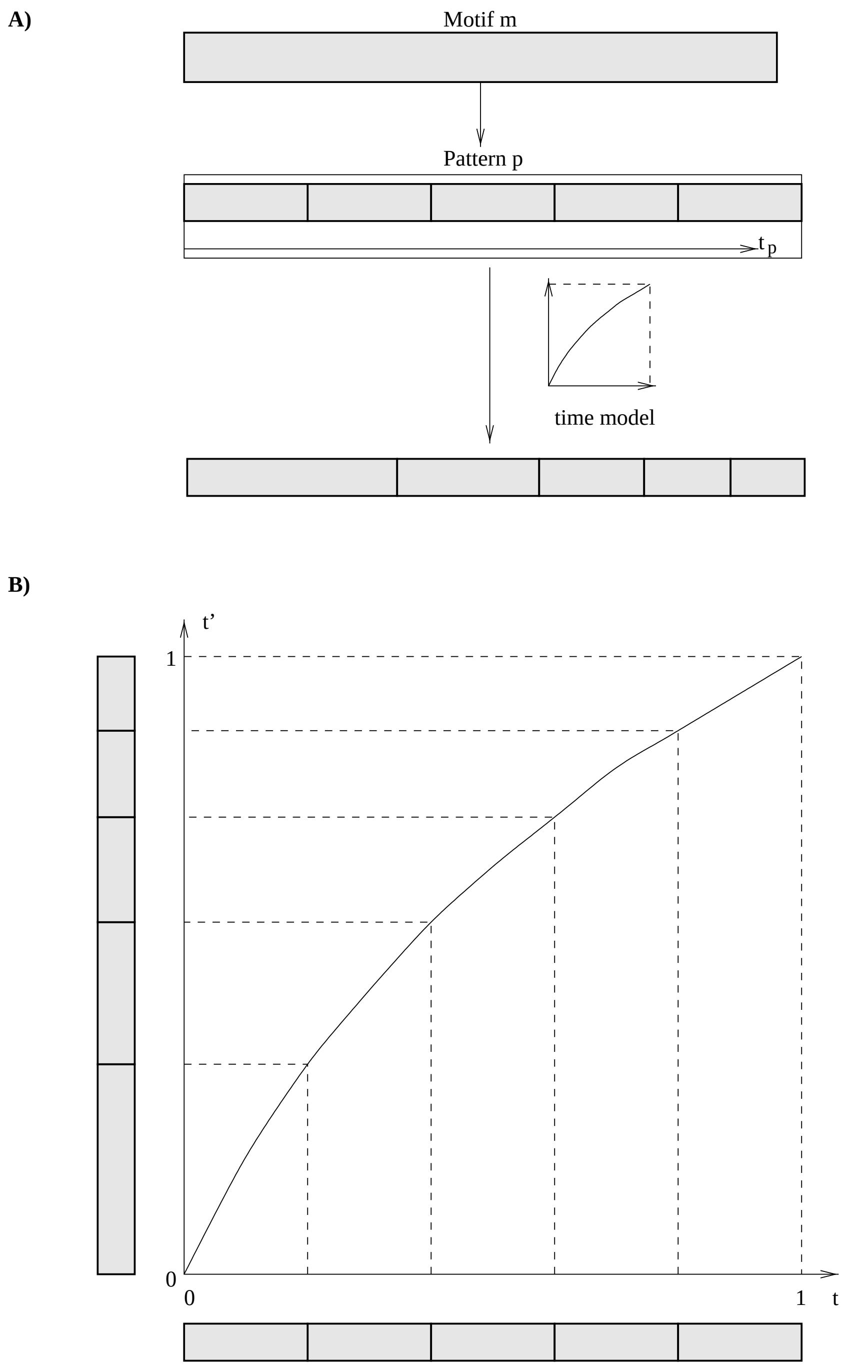

Figure 6.7: *The figure shows an example of a time model. A) On top is depicted the original pattern and motif as it is used in the composition. Below is shown the activities with their final durations after scheduling. B) The start times and the durations of the activities are mapped using the time model.*

- Time warping can be used to change the speed of execution of a composition. This change of speed does not have to be constant, which would be quite trivial, but can vary during the performance. This tempo change can be used to introduce expressiveness, but also to synchronize the playback with a performer.

The issue of time deformation can be seen more generally in the light of the representation of continuous time. Of course, no such thing as continuous time exists in digital systems. However, we have defined our control functions such that their value can be requested at any time value (section 5.3.1). It is in this fashion that we interpret the notion of "continuous" time functions in this text. Our proposition for the manipulation and integration of continuous time is a combination of Honing & Desain's generalized time functions [Hon93] and time warping, in particular Dannenberg's work [Dan97]. Other works relevant to this discussion are those by Rodet & Cointe [RC84], and Anderson & Kuivila [AK89].

To represent

- continuous control functions within compositional structures,
- non linear time relations between "local time" and "absolute time",
- a rich time model during the execution of a piece,

we will introduce

- a layered, hierarchical, and continuous representation of time,
- time models to warp time from one layer onto the next layer of this hierarchy, and
- time contexts to encapsulate the time information of an activity use during runtime.

The layered time models coincides with the structural hierarchy described by patterns and motifs. This rich time representation is used to transform the start times and durations of activities and to describe non-linear dependencies of control functions on absolute time.

To the hierarchy of motifs and patterns will we attach a hierarchy of time models. Consider figure 6.7. Motif m subscribes to pattern p. When motif m requests its contents, p clones its contained activities and organizes them according to m's duration. We will introduce a virtual time axis t_p for pattern p. In reality p has no such time axis. This axis only exists when p organizes the activities for a particular motif. In that case the length of the time axis is the duration of the motif. Once the activities are organized, their start times and the durations are mapped from p's virtual time axis onto m's time axis. For this mapping pattern p uses a *time model*. The example in figure 6.7 depicts a logarithmic time model. This time model provokes an *accelerando*. Time models are defined as follows:

Definition 5 (Time model) *A time model tm is a bijection that maps a value t onto a value t': $tm : [0,1] \rightarrow [0,1] : t' = tm(t)$.* **TM** *denotes the set of time models. A special time model is the linear model, $tm_{linear} : t' = t$.*

Time models are not only used to adjust the start position and durations of the activities. With every activity we associate a local time axis (see figure 6.8). Motif m has a local time axis t_m, and activity a has a local time axis t_a. These two axes stand in some relationship to each other. This relationship is described by the time model of the pattern p that contains a. How this relationship is articulated is shown in the figure. For the construction of this relationship we will rely on p's virtual time axis t_p. The mapping of the time axis of activity a onto the time axis of motif m requires the time model of the pattern p and the offset and duration of a. When we let the time grow linearly on the time axis of a, it will grow logarithmically on m's time axis. Vice versa, when the time advances linearly on axis t_m, t_a grows exponentially.

When the activities are scheduled, they will provoke the creation of new synthesis processes (see also figure 6.6). These synthesis processes and the control functions that steer them must experience the same time dependencies as the activities in the composition. We must therefore pass all the time information and the context in which the activity finds itself in the composition to the synthesis processes and their control functions. We will group this information in a structure called *time context* (Fig. 6.9). This time context ensures the correct conversion from the real time to the local time of the synthesis process.

Consider figure 6.10. An activity $a1$ starts at a time s_{a1} and has a duration d_{a1}. Both s_{a1} and d_{a1} are measured on the time axis of a_1's pattern $p2$. When a_1

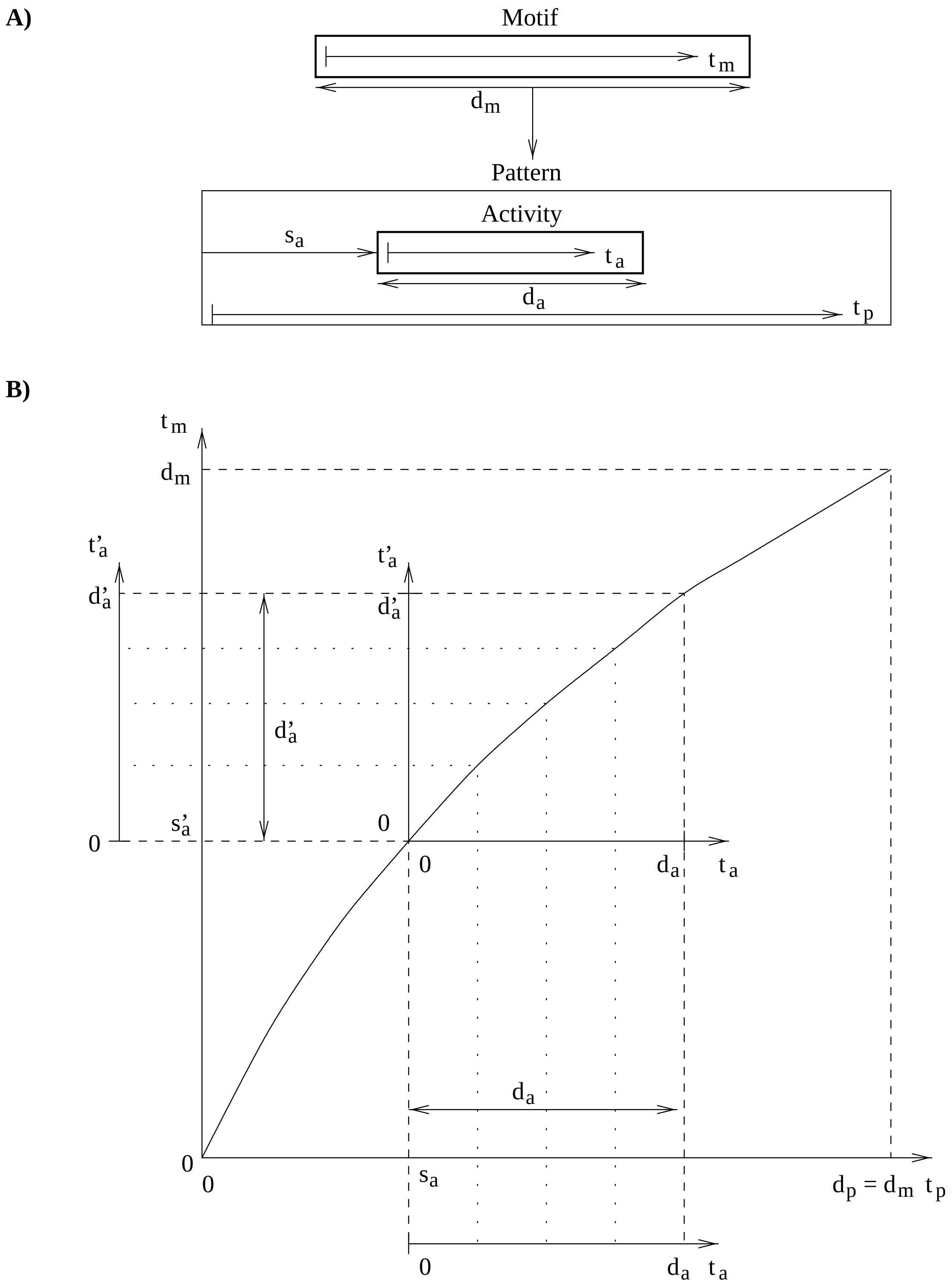

Figure 6.8: *Figure A) displays a simple organization. Motif m and activity a have a local time axis. Both axes stand in some relationship to each other. This relationship is described by the time model of pattern p. B) Time models describe the mapping of the time axis of an activity onto the time axis of a motif.*

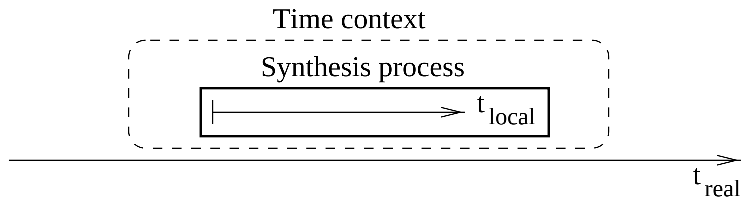

Figure 6.9: *Any synthesis process created by an activity is embedded in a time context. This time context describes the time dependencies, as defined by the composition, between the local time axis of the synthesis process and the real time.*

is performed, it experiences a time t_{a1}. Motif m_2 subscribes to the organization of pattern p_2 and is included into the pattern p_1. It has a start time s_{m2} and duration d_{m2} measured on the time axis of p_1. When motif m_2 is performed, it experiences a time t_{m2}. Motif m_1 subscribes to the organization of pattern p_1 and has a duration d_{m1}. When the user request the performance of motif m_1 at some time s_{m1}, activity a_1 will be scheduled at its appropriate time. The mapping of the time experienced by the activity a_1 onto the real time dictated by the system depends on the start times of a_1, m_2, and m_1, and the time models of p_2 and p_1. The mapping of t_{a1} onto t_{m2} is described by the time model of $p2$. The mapping of t_{m2} onto t_{m1} is described by the time model of $p1$. A synthesis process created by the activity a_1 must be able to reconstruct this hierarchy of time dependencies during the performance. For this purpose, we chain the necessary data into a series of time contexts.

Definition 6 (Time context) *A time context c is a quadruple $< s, d, tm, c' >$. $s \in \mathbf{R}_+$ indicates an offset time, $d \in \mathbf{R}_+$ a duration, $tm \in \mathbf{TM}$ a time model. c' is the parent time context of c.*

In the case of activity a_1 in figure 6.10 the context c is equal to $< s_{a1}, d_{a1}, tm_{p2}, c' >$. c' is the parent time context of a_1, i.e. the time context of motif m_2.

We can now fully specify the interfaces of the synthesizer object and of the activities. The add method of the synthesizer expects three arguments:

1. an identifier for the synthesis process (see section 5.3.2),
2. the time context of the synthesis process,
3. the synthesis process.

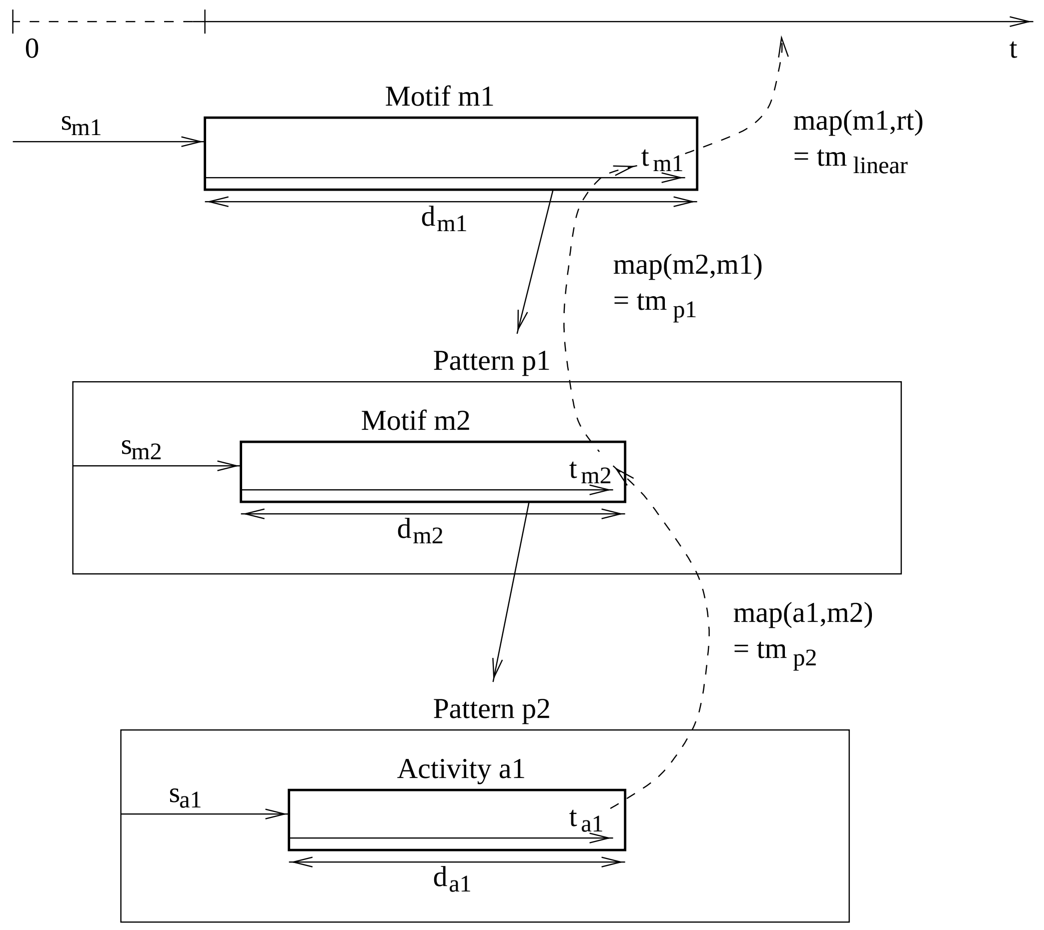

Figure 6.10: *The construction of time contexts*

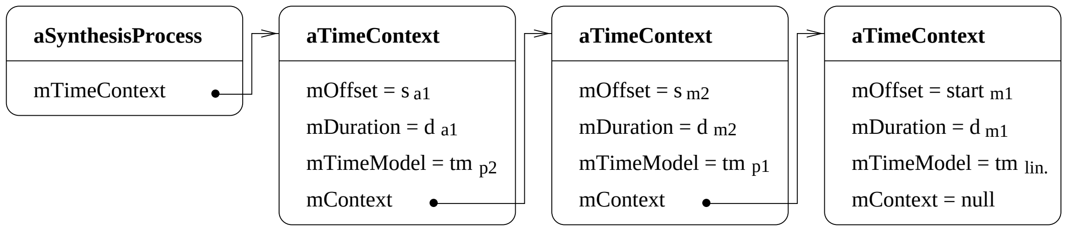

Figure 6.11: *Any synthesis process created by an activity refers to a linked list of time contexts. The synthesizer passes the synthesis process the absolute time. Using this time and the contexts, the process can calculate the time on any level of its parent's contexts.*

When activity a_1 is scheduled, its start program p_s is invoked. The first arguments passed to p_s is the time context c of a. c holds a reference to its parent time context, and so forth. The argument passed to p_s is thus a chained list of time contexts (Fig. 6.11). p_s creates a new synthesis process sp and adds it to the synthesizer passing along the time context. When the synthesizer calls the synthesis process sp it converts the absolute time indicated by the system to the local time of the synthesis process using sp's time context. When the synthesizer calls the synthesis process to produce the sound, it gives as parameters the synthesis buffer, the local time, and the time context. sp calls upon any control function with its local time and, again, the time context. Using this local time and time context, the control functions can calculate the time on any superior time level of the composition. This possibility will be used in the example of amplitude curves, portamento, and vibrato discussed in section 6.7. First, we take a closer look at activities used for sound production.

6.6 The case of sound activities

In this section we will discuss activities in the particular case of sound synthesis. We call them *sound activities.* Like any activity they have a start time s, a duration d, a start program p_s, and an end program p_e. A sound activity represents a sound that starts playing at time s and lasts for d time. Note that it can be any sound: short, long, pure, complex, simple, composed, etc...

In this section we will make the following assumptions about the sound activities:

- they have one parameter that describes the frequency. This frequency is measured in Hertz.
- they have one parameter that describes amplitude. The amplitude is measured on a linear scale from 0 to 1.
- their start program schedules a synthesis process, their stop program kills the same synthesis process.
- they have an associated synthesis voice. The synthesis voice is charged with the creation of the synthesis process.

The start program of an activity takes two steps. First, it creates the synthesis process by calling its associated synthesis voice. It passes the frequency, the amplitude, and any additional parameters to the synthesis voice (see section 5.3.3). Second, it sends an add-event to add the synthesis process to the synthesizer. The end program of a sound activity kills the synthesis process.

We include a remark here. Initially we accepted *timbre* as the essential concept to describe a sound and considered the classical notions of pitch and intensity as attributes of the timbre. We therefore did not want to "hard code" frequency and amplitude in our environment. Studies in the field of musical perception and cognition seem to suggest, however, that the pitch and intensity of sound do occupy a special role in the sound perception (see for example [SD93] and [PPK99]). In addition, not introducing frequency and amplitude would have made most musical transformations cumbersome and loaded the problem of "timbre representation" on our backs. We therefore accept the frequency and amplitude as fundamental parameters for sound. Sounds that do not fall in this category (noises of all trades, for example) get this interface for free. They benefit from it, if possible, and can use, for example, the amplitude parameter while neglecting the frequency parameter.

We also define the interface `PitchedSound`. This interface defines the methods `getFreq`, `setFreq`, `getAmp`, and `setAmp`. Any object that can handle frequency and amplitude implements this interface and can participate in musical transformations. We use this interface to define sound patterns and sound motifs. They specialize the `Pattern` and `Motif` class, respectively. An amplitude envelope and frequency curve can be set for both objects. Associated modifiers can use these curves to change the pattern. We will give examples in the next section.

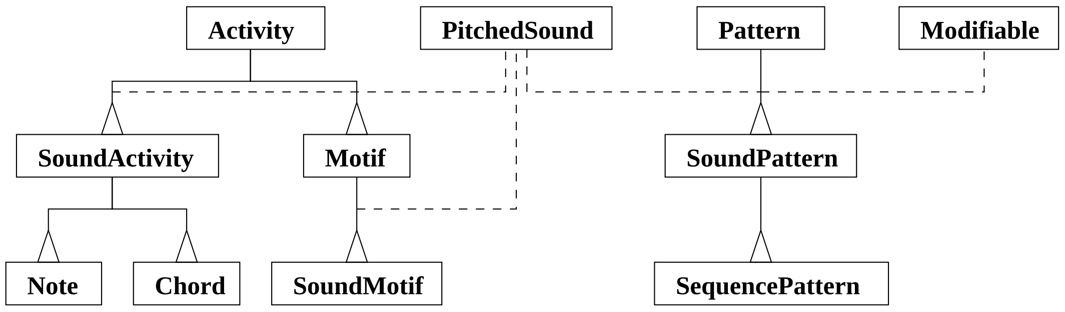

Figure 6.12: *The classes and inheritance tree for sound activities. We used a solid line for class inheritance, and a dashed line for Java's interface inheritance.*

We have also defined the class `Note` and `Chord` for convenience. Both classes inherit from `SoundActivity`. The note class allows the creation of sound activities with the following parameters:

- A start location and duration.
- A synthesis voice.
- A constant pitch value in midi-cents[1]. This value is converted into a constant controller returning the value in Hertz.
- An amplitude value in decibels. This value is converted into a constant controller returning the value in linear amplitude.

The chord class is very similar, except that it accepts a list of pitch and amplitude values. For reasons of efficiency we choose to model a chord as a single activity instead of a pattern. An overview of the musical classes is given in figure 6.12. In the next section we will give some examples of modifiers in the context of sound activities.

[1]Midi-cents is a combination of a MIDI key number (between 0 and 127, with 60 for the middle C) and a cents value. The cents unit divides a semi-tone interval in 100 equal parts. Thus a midi-cents value of 6050 indicates 50 cents (or a quarter-tone) above the middle C (key number 60).

6.7 Some examples of the manipulation of sound activities

In chapter 3 we discussed several problems among which the stretching and vibrato problem. We will now consider how we solve these issues in the proposed environment. The following discussion provides some concrete examples of modifiers.

6.7.1 Accelerando

An accelerando is the continuous speeding up of the performance. We have discussed accelerando briefly in the discussion on time models. The following lines show how to apply a time model to a sequence of four notes. In this example we see also how breakpoint functions discussed in section 5.3.1 can be used as time models. The time model we define here is the one depicted in figure 6.7 earlier on.

```
(define s (sequence 2))
(sequence-add (note 0 1 vibra-voice 6000 -10))
(sequence-add (note 0 1 vibra-voice 6200 -10))
(sequence-add (note 0 1 vibra-voice 6500 -10))
(sequence-add (note 0 1 vibra-voice 6400 -10))

(define t (bpf 1 '((0 0) (0.1 0.19) (0.2 0.34) (0.3 0.46)
                   (0.4 0.57) (0.5 0.66) (0.6 0.74) (0.7 0.82)
                   (0.8 0.88) (0.9 0.94) (1.0 1.0))))

(set-time-model! s t)

(play s)
```

We can apply a time manipulation, such as a stretch, to the sequence defined above. Since the time-model does not depend on the duration of the motif, the contained notes are spread correctly over the length of the sequence.

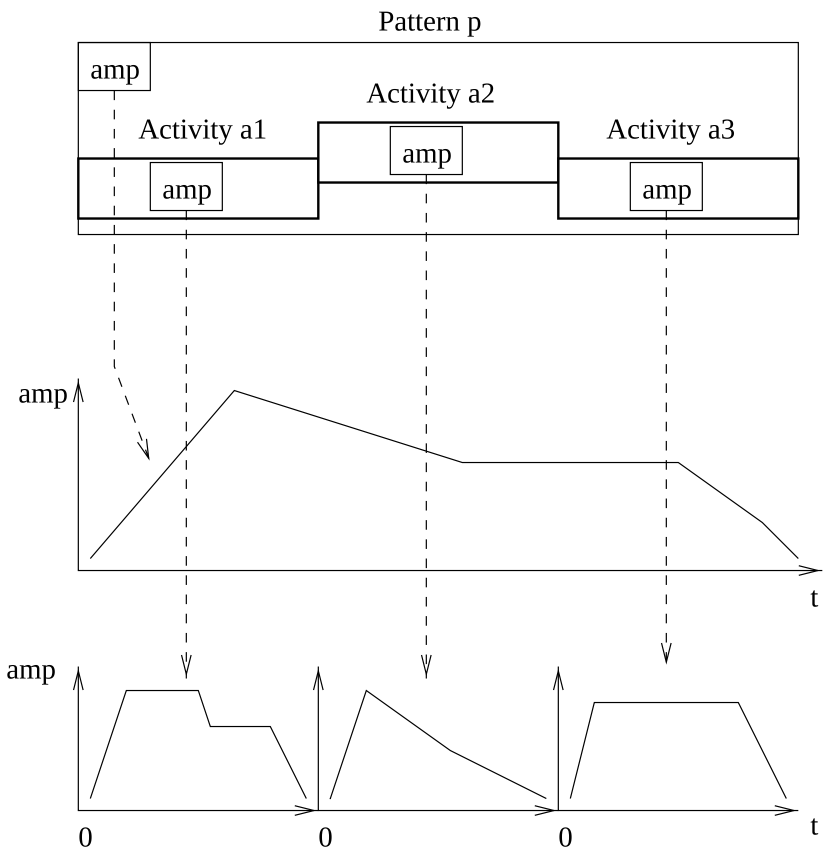

Figure 6.13: *Defining an amplitude envelope for a pattern.*

6.7.2 Amplitude envelopes

In music, it is common to indicate an evolution of the dynamics for a section in the composition. We will consider the case of a sequence of notes. Figure 6.13 displays a pattern p with three sound activities a_1 to a_3. Each of the activities has its local amplitude envelope. A sequence pattern implements the interface defines by `PitchedSound`. The user can therefore set a control function indicating the amplitude curve for the sequence. This envelope should be applied to the contained activities.

To solve this problem we define an amplitude envelope modifier and an amplitude envelope control function. The modifier is called after the pattern has organized the activities for a motif. It collects all the amplitude control functions of the individual sound activities in the pattern and replaces them with one global amplitude control function. This global control function keeps a reference to the initial control functions of the activities and to the amplitude function of

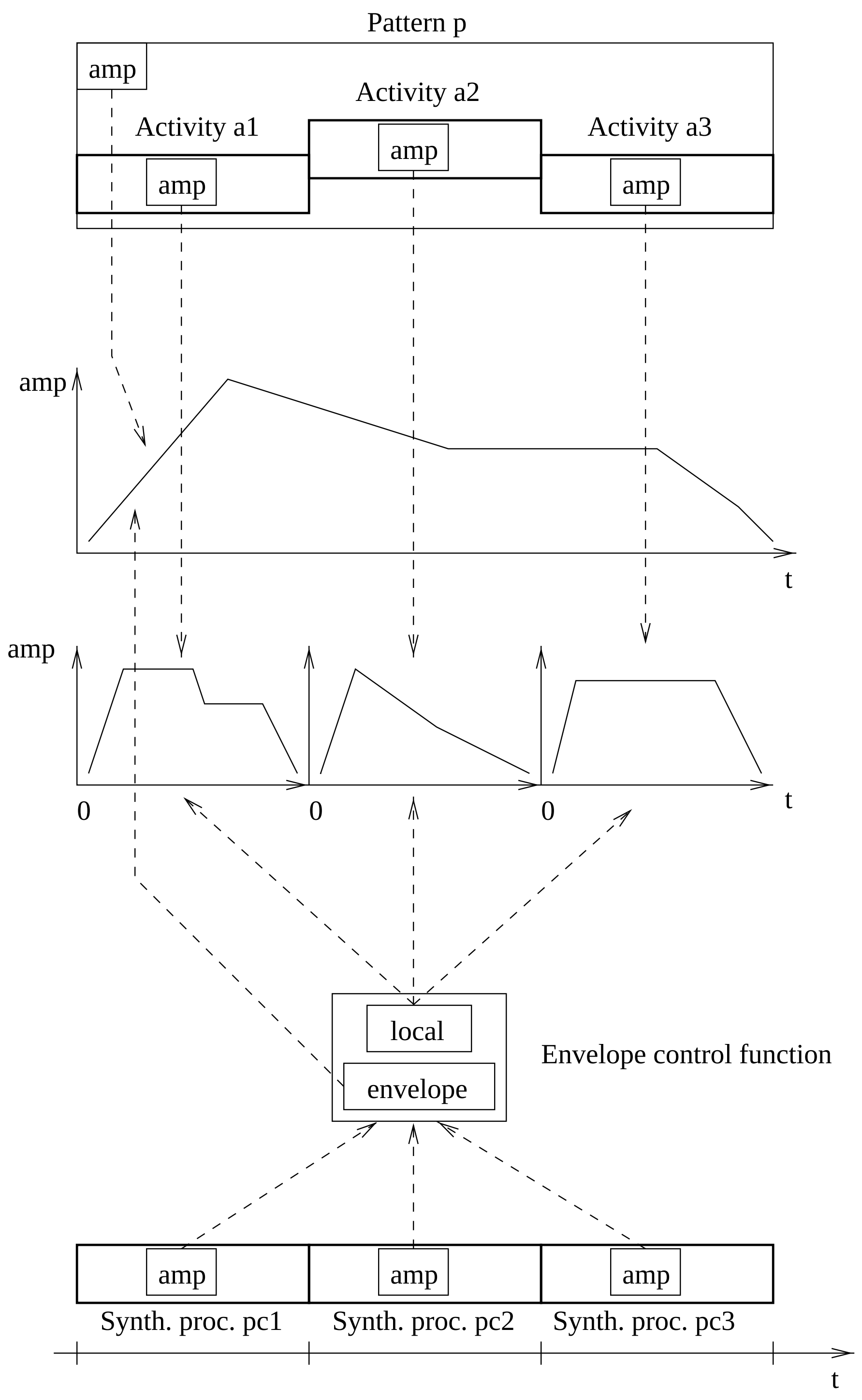

Figure 6.14: *The amplitude envelope modifier*

the pattern. The local amplitude controllers of the activities are replaced with the global controller. When an activity a_i of the pattern is scheduled it creates a synthesis process sp_i. This synthesis process calls the global controller to obtain its amplitude value. To calculate the amplitude value the global controller first calls the initial amplitude function of the activity a_i then calls the amplitude function of the pattern p and finally multiplies the two values. Note that the values of both amplitude functions of a_i and p must be called on distinct time axes. Since the global controller receives the time context of the synthesis process sp_i as parameter, it can convert the local time of a_i to the global time of p. In this solution we retain all the initial information of the individual activities. Inspection of the control functions and modifications at runtime are still possible. The following lines of Scheme code shows how to create and insert an amplitude modifier into the sequence used earlier on:

```
(define envelop (bpf 1 '((0 0) (0.25 1) (0.75 1) (1 0))))
(set-amp! s envelop)
(motif-set-modifier! s (amplitude-modifier))
```

6.7.3 Portamento and vibrato

A similar problem is portamento: a melody of notes with distinct pitches is performed in one continuous gesture. The rupture in pitch between two consecutive notes is smoothened: one note is “carried” to the next one. We solve this problem with the portamento modifier and the portamento control function (Fig. 6.15). The portamento modifier collects the frequency control functions of the individual activities in the pattern and replaces them with a new portamento control function. This control function returns the value of the activity's frequency control function, or an interpolated value on the edges between two activities.

```
(motif-set-modifier! s (portamento-modifier))
```

Vibrato is the modulation of the pitch. The vibrato in a performance is, in general, coherent over a section over the music. The modulation is continuous when the performer moves from one note to the other. Furthermore, the rate of the modulation is independent from other time variations such as accelerando,

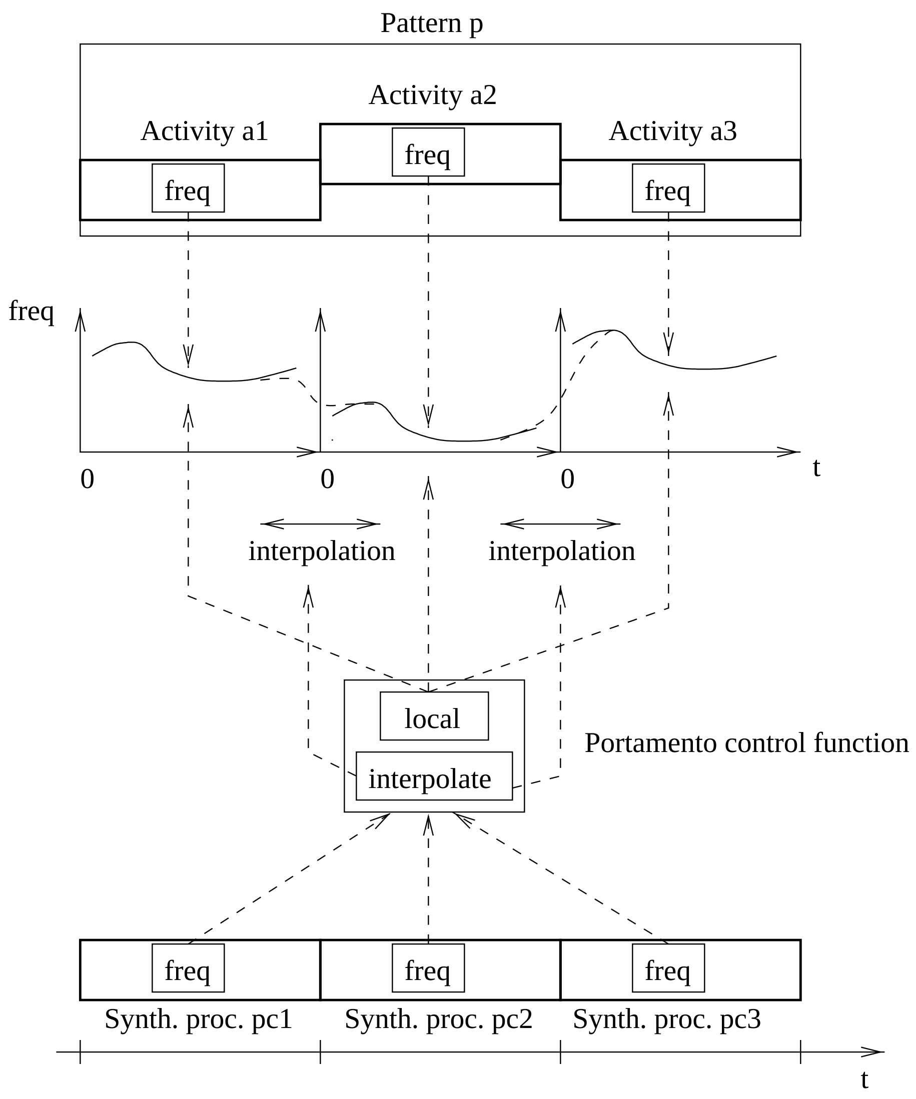

Figure 6.15: *The portamento modifier*

etc... The vibrato in the following example is characterized by a constant frequency (in Hertz) and amplitude (as a percentage of the initial frequency). To apply a vibrato modifier to the sequence, the next line of code is entered:

```
(motif-set-modifier! s (vibrato-modifier 6.0 0.02))
```

Of course, the amplitude envelopes and the portamento scale accordingly when an accelerando time model is set for the pattern.

6.8 Discussion

In this chapter we have discussed how multimedia pieces, and in particular music pieces, can be structured. The basic element of a composition is called an activity. It is defined very generally as an object that has a location and duration in time. How the "contents" of the activity is played and stopped is described by the start and stop program of the activity. In the case of sound activities the start program creates and adds one or more synthesis processes with the help of a synthesis voice. The stop program simply kills the synthesis process.

Composition is possible in various ways:

- Composition algorithms can be expressed in the Scheme language. When the activity is called for the scheduling the algorithm constructs its content with the help of the algorithm.

- A structure can be defined progressively thru the use of a pattern. A pattern groups a number of activities and defines the relations between them. After every modification to a contained activity the relations are verified. A pattern is not used immediately into a composition. They are used thru the intermediary motifs. Motifs are a "handle" to a pattern and define the context in which the pattern should be organized.

- The organization of activities whose duration is determined during runtime by user interactions is possible with the use of "causal" relations. The start and stop programs of related activities are recombined to create the desired dependencies.

The synthesis processes that are created have a local time space. The local time is mapped onto the "absolute" time thru the use of a hierarchy of time models. These time models can be defined in the composition. The hierarchy of the time models corresponds the structural hierarchy of patterns and motifs. All the time information necessary for the time conversions is passed on to the synthesis process in an object called "time context." This rich, layered time model provides the means for complex, non-linear time deformations.

In future projects we would like to extend the current model. First, we want to explore the technique of local constraint propagation for use in the description of patterns. Second, it would be interesting to include an activity into several patterns simultaneously. This is more than just a feature. Musical organization has always been characterized by a complex network of relations. The type of relations are various: structural, harmonic, dynamic, timbral, and so on. Several layers of organization are always in play with the piece. It would be interesting, both from a compositional as from a computer science point of view, to develop a representation scheme for this complex problem.

Thus far we have discussed the architecture of the environment and its principal classes. We have also given examples of the interface to the Scheme interpreter. We have seen how to define synthesis techniques and how to construct music pieces. One question remains: Can such a complex environment really work in real-time? We will analyze this in the next chapter.

Chapter 7

Analysis of the real-time performance of the environment

Thus far we have discussed the architecture of the environment and its major classes for synthesis and composition. We have seen that the environment can be used both for the composition and for the synthesis of a piece. Much of the functionality of the architecture lies in the use of an embedded Scheme interpreter. In addition, we propose to use the environment in an interactive set-up. In that case the system has to run in real-time. However, the interpreter and the synthesis task have quite different requirements and time behavior. On the one hand, the synthesis task is a real-time task. It requires a very controllable usage of the system resources in order to guarantee a correct behavior. On the other hand, the interpreter is very erratic. Because of the complex network of dependencies between the objects in the environment, the interpreter typically delegates the task of freeing unused objects to a garbage collector. This makes the runtime behavior of the interpreter very difficult to predict. Since both tasks run in the same space, the interpreter may interfere with the synthesizer causing its deadlines to be missed.

With our current prototype implementation, we offer soft real-time performance. That is, the synthesizer runs fast enough for direct output, but an occasional interruption may occur and cannot be completely avoided. This is sufficient for studio work. However, if we want the use such an integrated environment in critical situations such as concerts and music installations it must be capable of hard real-time performance.

In this chapter we will investigate two points:

- We will analyze whether hard real-time is possible *at all* in such an integrated environment. If this is not possible, we will be forced to define a new architecture for use in critical situations. If it is possible, we will point out what is left to be done to make the current implementation real-time.
- Independent of whether hard real-time is possible or not, we can list a number of design decisions that will limit the influence of the garbage collector on the real-time task.

We start in the next section with a short description of the interaction between the main components of the system in runtime. In this overview we will take the opportunity to highlight the different stages in the life of a synthesis process. Then we will take a look at the particular case of the interaction with the garbage collector (Section 7.2). We estimate the worst case execution time of the synthesizer (Section 7.3). Finally, we discuss the real-time possibilities of the architecture (Section 7.4).

7.1 The dynamic view of the architecture

Figure 7.1 shows an example of an interaction between the main components of the system in runtime. We adopted the notation used in [GHJV95] to which we superposed the activity of the threads. The time flows from top to bottom. The vertical rectangles show when an object is active. A method call is represented by an arrow from the calling to the called object. The label above the arrow indicates the method name. We have tried to visualize the switching between the three threads: when a vertical rectangle is shaded gray, the thread in which the object is called is suspended. On the right hand side is indicated which of the threads is active.

In figure 7.1 the event thread calls a program to handle an add-event. The application of the event's program provokes the creation of a synthesis process. To store the new object storage space is requested from the heap. After the creation, the synthesizer is requested to add the new synthesis process to the list of active processes. From this point on, the synthesis process is said to be active. The life of a synthesis process knows four life stages. When it is just created its

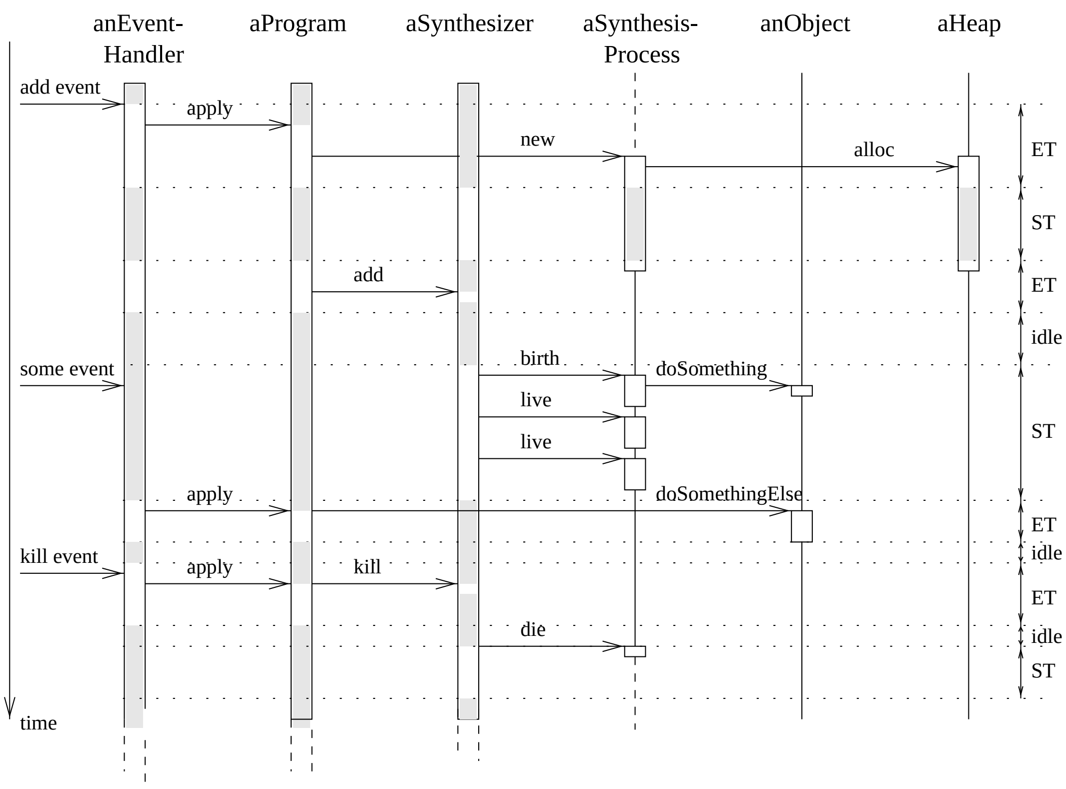

Figure 7.1: *The interaction diagram shows the interaction of the main components of the system in time (see text for a more detailed explication.) (ET = Event thread, ST = Synthesis thread.)*

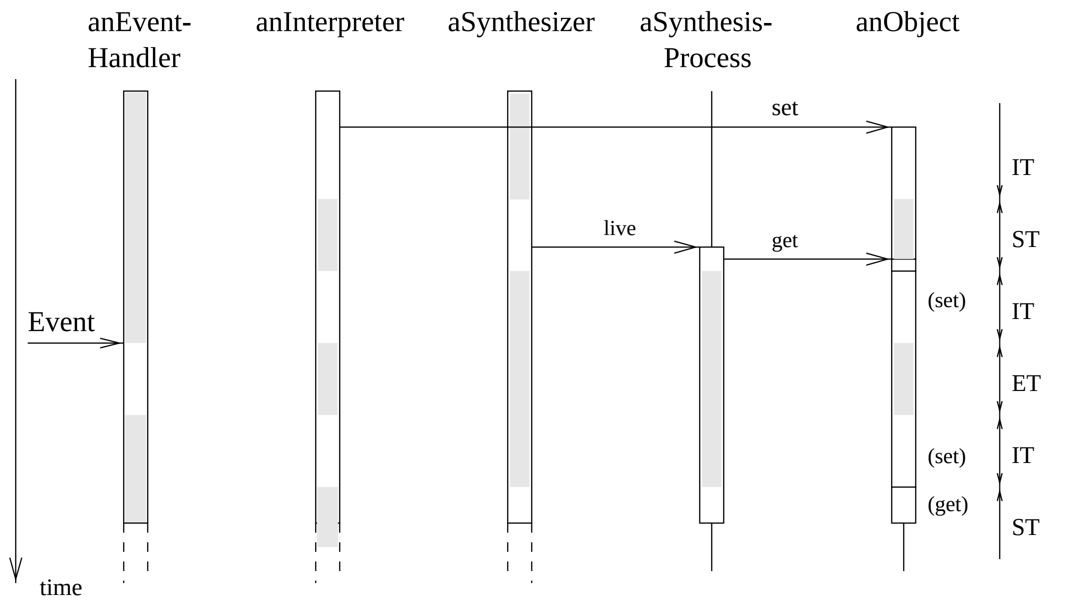

Figure 7.2: *Priority inversion (IT = Interpreter thread, ET = event thread, ST = synthesis thread)*

state is called *embryo*. When it is added to the synthesizer it is called once with the message `birth`. The `birth` method allows the process to do specific work when it is called for the first time (a fade-in, for example). The process is then said to be *alive* and will subsequently receive the message `live`. A `kill` event (the third event in the figure) will cause a currently active process to be removed from the synthesizer. The process is not immediately eliminated; its state is set to *dying*, and it gets another chance to finish its song. It is called once more with the message `die` in which it can do a fade-out, for example. In the last stage, called *heaven*, the process is removed from the array in the synthesis thread.

We can see in the figure that the event thread is constantly interrupted by the synthesis thread. Since the synthesis is more important than the event handling, it has priority. The synthesis is a hard real-time task and should never be delayed since this would introduce audible clicks in the sound output. The synthesis thread has therefore the highest priority. The processing of incoming events is a soft real-time task. The events should be handled as fast as we can. If an event specifies the start of a new sound, for example, the delay should be less than 30 msec. However, it's better to handle the event a little to late (say 50 msec) than not at all. This task has a high priority but lower than the synthesis task. The interaction with the user thru the Scheme shell or any other interface is not, strictly speaking, a real-time task. Granted, to have a good interaction, responses to user input should be fast. Nevertheless, this task has the lowest priority of the three.

In most cases the synthesis processes will depend on other objects. For example, an oscillator will depend on a controller object indicating the frequency of the oscillator. The oscillator retrieves the value of the controller at regular time intervals. If the controller object is defined in the Scheme environment, incoming events and the subsequent application of programs can modify the state of the controller. This makes the controller a critical zone (see section 1.3) since it is called by several concurrent tasks. Access to its shared data must be synchronized. This is detailed next. Consider the flow of events shown in figure 7.2. The Scheme interpreter sends an object the message to change its value. The method `set` is a critical region and the object protects its shared data with a monitor. At that point the synthesis thread wakes up, suspends the Scheme thread (lowest priority), and calls the synthesis process. The synthesis process sends the object the message `get` to obtain its value. Since the interpreter thread still possesses the monitor (the `set` method), the synthesis thread will block when it tries to enter the object's monitor. As a result the interpreter thread is resumed again. Before it has had the time to leaves the critical region an event arrives. Now, the event thread wakes up and suspends the interpreter thread (still lower priority). At this point the synthesizer thread must wait for the event thread to finish handling the event. This event handling may take very long and the synthesis thread may be delayed for an unpredictable amount of time. This is the problem of priority inversion we mentioned in section 1.3. It is a well known problem that is generally solved with the technique of priority inheritance. In the situation above, the interpreter thread would *inherit* the priority of the synthesis thread, in which case it cannot be suspended by the event thread. Despite this fact, the synthesis thread still has to wait for the interpreter thread to leave the critical zone. With priority inheritance, the maximum delay of the synthesizer is no longer unpredictable but related to the size of the critical zone. *However, if the computation in the critical zones is reduced to a minimum, these delays will be reduced to a minimum.* In the next section we continue the study of the interaction in the special case of the garbage collector.

7.2 The influence of a garbage collector

The system considered in this thesis integrates two subsystems that traditionally handle memory allocation and reclamation differently. On the one hand, the Scheme interpreter typically uses a garbage collector for the reclamation of unused memory space. On the other hand, real-time systems avoid dynamic memory management since it makes the behavior of the system hard to predict. We face three options for the memory management: an implicit management (garbage collection), an explicit management (hand-coded), or a hybrid management (partly explicit, partly implicit.)

Choosing implicit memory management places a garbage collector face to face to a real-time task. It will be difficult, a priori, to predict the system behavior and guarantee hard real-time conditions. However, if we choose to use explicit management we will run into problems to realize the Scheme interpreter. In such a dynamic, interpreted environment the relationships between the objects are complex and deciding when storage space can be released is tricky and error-prone. It may also necessitate a complete rewrite of existing interpreters to adjust to this solution. Furthermore, as shown by Nilsen [NG95] traditional allocation algorithms have bad worst case delay times, which does not make them a suitable option for hard real-time. If we choose a hybrid system the interpreter would rely on a garbage collector to clean the heap; the real-time task would manage its heap explicitly. It might be an intermediate solution but does introduce new problems. Both heaps must cooperate to resolve references across the boundaries of the two heaps. This coordination might be more unpredictable and take more time than an optimized garbage collector. Furthermore, the system looses flexibility if the objects in the environment cannot be fully shared between the two tasks.

Since we have chosen to develop in the Java language we will rely on its implicit memory management. We will examine the influence of the garbage collector by studying the possible cases of interaction between the two subsystems. Consider figure 7.3, for a start. The synthesis process calls an object; the object in its turn allocates storage space in the heap. The heap, however, does not have enough space available to satisfy the request and is forced to scan the heap for unused memory and collect the garbage. This collection will most probably take a fair amount time and the synthesis process will not be able to calculate its output value in time. The synthesis thread will not be able to deliver the output

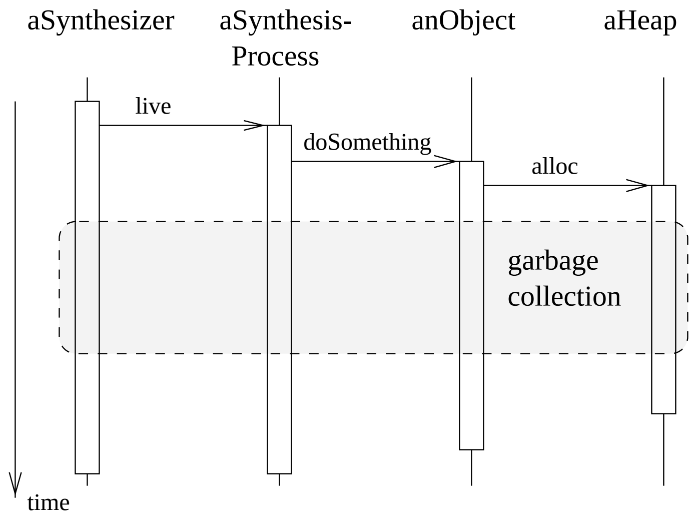

Figure 7.3: *Garbage collection during the synthesis will block the synthesis thread.*

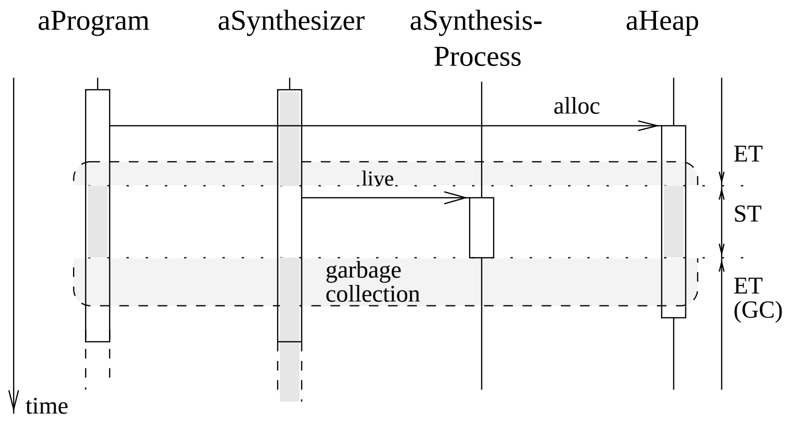

Figure 7.4: *Garbage collection during the event handling does not necessarily block the synthesis, if the collector is concurrent.*

in time to the audio device and will no longer be synchronized. *This situation must absolutely be avoided. Therefore, we impose the constraint that the synthesis processes and all the objects they call do not allocate new storage space during the synthesis.* All the methods called during the synthesis should only perform calculations or assign new values. They should not allocate storage space. This is not necessarily a big constraint: it is the design any developer of a signal processing technique would choose to optimize the algorithm. Obviously, the objects *can* allocate storage space, but only when they are called in the event or user thread. If this constraint is taken into consideration, the synthesis thread will never *directly* rely on the services offered by the garbage collector.

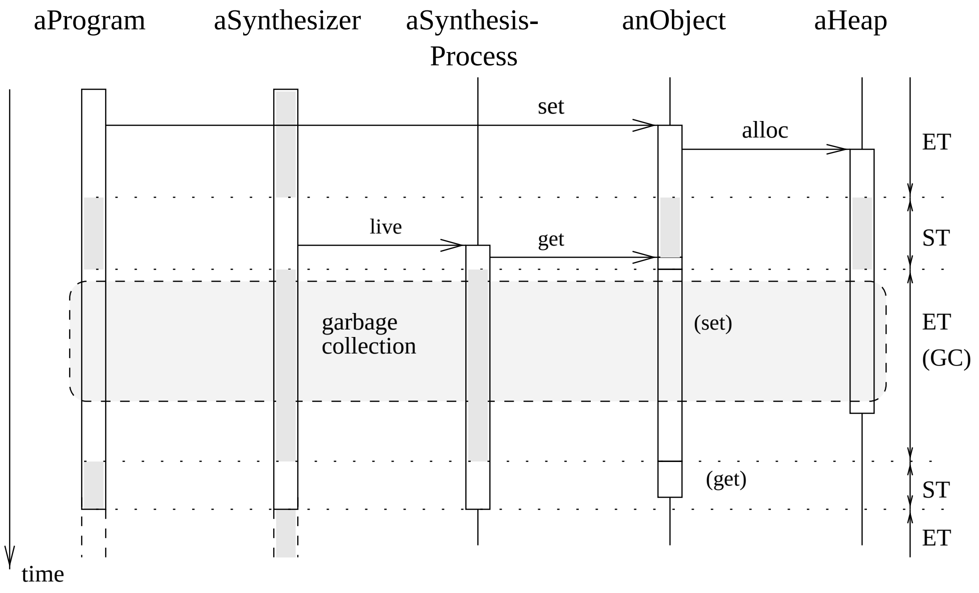

Figure 7.5: *Garbage collection in a critical region may provoke a priority inversion; the synthesis thread will be delayed.*

A seemingly disastrous situation might occur as shown in figure 7.4. A program, called from within the event thread, requires storage space from the heap. The heap does not have enough space available and performs a garbage collection. Again, this collection might take a lot of time. Sure enough, the synthesis thread will need to acquire the CPU before the collector finishes. *This situation does not raise any problems, if the garbage collector can work concurrently to the other threads.* In that case the thread allocating storage space is suspended; all other threads that do not allocate memory are not suspended. When the synthesis thread needs to come into play it simply suspends the garbage collector since the latter has a lower priority. This means that the garbage collection need not interrupt the synthesis in this case neither.

Problems can arise, however, if an object allocates storage space from within a critical zone (Figure 7.5). In this case a priority inversion might occur, and when the synthesis thread tries to access the object it will find itself waiting till the end of the garbage collection. To avoid this problem, *the objects on which synthesis processes rely should not allocate storage space inside a critical region.*

So far we looked at the interaction between the components of the system. However, the synthesizer itself must be able to keep its worst case delay bounded. We will consider this in the next section.

7.3 Protecting the system from overload

The system has to calculate an output value every τ seconds. If we consider sound synthesis using a buffer of 64 samples and a sampling rate of 44100 Hz, the value of τ is a little over 1.45 milliseconds. This time has to be divided over several functions, including the sound synthesis performed by the synthesis processes. If these functions together take more than τ time, the system will not be able to output its sound buffer in time, and clicks are introduced in the sound. This should be avoided at all costs. In this section we will examine how this short interval of time is used, and how we can prevent the system from overloading the CPU.

In pseudo code, the algorithm of the synthesizer to calculate one output looks like this:

```
synthesize one output {
  do initialization,
  call each of the synthesis process,
  write the output value
}
```

Let τ_s be the time needed by the system to calculate the output. τ_s is split into the following fractions:

- τ_0 - the time needed by the synthesis thread, even when no synthesis processes are active (to clean buffers, etc…).
- τ_i - the time needed by synthesis process sp_i to calculate its output.
- τ_{wr} - the time needed to write the output to the output device. This time depends mainly upon the operating system and the driver.
- τ_{sw} - the time needed to switch between the running thread and the synthesis thread. This time laps depends on the operating system.
- τ_m - we also include a margin to accommodate fluctuation in the previous times, and to guarantee a minimum time to the other threads (event handling and user interaction).

Let $\tau_{s,0} = \tau_0 + \tau_{wr} + \tau_{sw} + \tau_m$ be the time needed by the system when no synthesis processes are active, and let $\tau_s = \tau_{s,0} + \sum \tau_i$. To protect the system from overload τ_s must always be smaller than τ. If $\tau_{s,0} \geq \tau$ the system can not perform any real-time synthesis at all. Otherwise the system can accept synthesis processes as long as $\tau_s < \tau$. If $\tau_s > \tau$ the system is overloaded and can not assure the delivery of the output values in time. Furthermore, the synthesis thread might take up all the CPU time, in which case the other threads are completely blocked; the system will have to be interrupted brutally. This situation must be avoided, especially in concert situations. How can we protect the system from such an overload?

- The traditional approach for real-time systems is to analyze the source code statically. After a very careful examination the maximum calculation times of all the function calls are determined and the maximum execution times of the real-time tasks are established. Since we know the maximum time a task can take, we can guarantee a correct behavior. However, this technique is very static and cannot cope with the dynamic aspects of our system.

- Before a new synthesis process is added, the synthesizer negotiates its resource requirements ("quality of service"). The synthesizer asks the synthesis process the maximum CPU time it needs for its calculation. If the system has that much time available, the synthesis process is accepted. If not, the synthesis process is not added or is asked to lower its requirements.

The quality of service approach is clearly better suited for a dynamic environment. However, we still have a number of problems:

- How do we measure the time τ_i used by a synthesis process?

- How do we keep a synthesis process to its advertised CPU requirements?

We could perform a code analysis when a synthesis process is added. However, as we have seen, a synthesis process can be a complex network of objects. Moreover, this network can change dynamically. So it will be very hard, a priori, to measure the CPU needs of a synthesis process. Even if we could guess its execution time, how do we prevent it from taking more time?

Nilsen [Nil] proposes the following solution for real-time systems written in Java. When a real-time object is scheduled, the system negotiates its CPU and memory usage. If the system can not satisfy the object's initial requirements, it asks to lower its demands. When the object is accepted, the system examines the Java code of the object, and, in particular, the code inside the event handlers of the object. From this analysis, it deduces the maximum time the object needs to respond to an exception. If during the real-time execution the object takes more time than given, the system raises an exception that will be intercepted by the object's exception handler. Since the time needed to handle the exception is known, the system can continue within fixed time limits.

Transposing Nilsen's solution to our case, the synthesizer could raise an exception when the synthesis processes takes too much time. In addition, we could advice developers not to define exception handlers in the signal processing code of the synthesis processes. In this situation the exception will be caught by the synthesizer. We can then design this handler to take the appropriate measures to reassure a timely behavior. Experiments of this kind are left for future work.

Before we end this section we would like to remark that due to internal buffering of sound data by the sound driver, an occasional delay in the sound output need not be catastrophic. However, in our argumentation we did not include this feature for several reasons. First, an occasional delay can be tolerated, but, in the average, the above discussion is valid. Second, the internal buffering should be reduced to a minimum since it introduces delays. Ideally, for real-time systems, no buffering is done. In the light of the previous sections and with chapter 1 in the background we can now argument the real-time possibilities of the system.

7.4 Design strategies and real-time possibilities of the environment

We will analyze whether hard real-time is possible *at all* in an integrated environment combining a real-time synthesizer and a high-level Scheme interpreter. If this is not possible, we will be forced to define a new architecture for use in critical situations. If it is possible, we will point out what is left to be done to make the current implementation real-time.

In reality, we have several problems at hand:

- The real-time task competes for the CPU with other tasks,
- The real-time task must synchronize with a garbage collector,
- The real-time task has a very dynamic load of its own.

As we have seen in section 1.3, real-time, multi-threaded applications have been studied for long. With the rate monotonic scheduling a correct assertion can be made about the application meeting its deadlines. Still, the synchronization between the threads can cause a priority inversion. This problem can be reduced thru the technique of priority inheritance. The synthesizer thread may still have to wait on a low priority thread when accessing a shared resource. However, a careful design of the critical zones should reduce the worst case delays of this synchronization.

The second problem is the synchronization between the garbage collector and the synthesizer. We gave an overview of the subject in section 1.2. We have seen that Wilson & Johnstone [WJ93] propose a hard real-time garbage collector. The collector uses an incremental, non-copying implicit strategy and uses an efficient write barrier to synchronize the collector and the mutator. In addition, in section 7.2 we have imposed upon the synthesis processes the constraint that they cannot allocate any objects. Synchronization between the real-time synthesizer and the garbage collector is then only needed in the following situations:

- Root scanning: The root scanning is an atomic operation. All threads are suspended.
- Write barrier: Additional instruction may have to be executed at every pointer assignment and add a fixed time penalty to this assignment.

The most restrictive requirement may be the root scanning. If this action takes too much time, it reduces our chances on hard real-time. The only other synchronization is the write barrier which can be implemented very efficiently and can be realized within fixed time bounds. The real-time task can perfectly run concurrent to the garbage collection and suspend the collector thread when needed. A part from the root scanning, the synthesizer will not be blocked by the collector. A performance price has to be payed for the write barrier. Since the synthesizer thread will not move a lot of pointers around, the penalty is fairly small.

The last point on the list is the worst case execution time of the synthesizer itself. We have seen that because the load of the synthesis processes varies dynamically, it is very hard to estimate the execution time. We have mentioned the use of exceptions handling to impose a time bound delay but leave this approach for future work. In critical situations, we can also retreat from this very dynamic load and fix the number and the algorithms of the synthesis processes.

Our conclusion is that it should be possible to use the environment in real-time situations. However, the currently available Java implementation cannot provide such stringent guarantees for the following reasons:

- They should be available on hard real-time systems.
- The Java threads should be mapped onto the native system threads and benefit from the real-time scheduling.
- The garbage collector should be designed for real-time.

It would be an interesting project to port a freely available Java virtual machine implementation on top of a real-time system, the real-time Mach kernel for example. Taking great care in the design of the garbage collector, we could then take accurate measures of the real-time performance. This project is left for future work :-)

It is interesting to note, however, that the Java language was from the start designed with real-time applications in mind [GM95]. Late 1998 Sun Microsystems announced they intent to develop the real-time extensions for the Java platform. Java can count on a wide acceptance, both from academic and private institutes. We foresee that the interest in and the research on real-time Java applications will grow. When Java is ready for real-time and virtual machines will be embedded in electronic devices of all trades, we will have an integrated music system available that can run on your-every-day-yet-sophisticated washing machine. Despite the soft real-time performance of our current prototype, test with this environment are very satisfying. This suggests that the system is very much usable in non-critical situations.

Conclusion and future work

In this thesis we have proposed an integrated environment for music composition and synthesis. Our system integrates a high-level Scheme interpreter, a real-time synthesizer, and a rich framework for temporal composition and sound synthesis. These are some of the reasons why we found it necessary to develop such an environment:

- The integration of timbre and the control of sound synthesis into a music composition: In our environment the description and control of the sound synthesis becomes an integral part of the composition. There is no longer a need for two programs: one for the composition and one for the synthesis. This integration allows the manipulation of timbre as one of the musical elements in a piece.

- A rich time model: Most synthesizers have a very limited notion of time – in the general case, they only know the current time. In our environment, however, the synthesis processes and controllers have the full knowledge of the time context of the composition. This knowledge is necessary if we want time manipulations, such as a stretch, to behave correctly in all situations.

- Real-time playback: We want music pieces to be played in real-time from within the composition environment. This should not only make the work process more efficient, but should also allow instant feedback and control of the composition.

- A fully dynamic environment: Most synthesizers proceed in two steps: programming the synthesis patch and executing it. We want a fully dynamic environment in which the composer can act upon the synthesis during the execution. This runtime programming is important to make adjustments to a composition that can only be verified during runtime. Interactive pieces can use this feature to manipulate the piece being played following user actions.

- An open system: We cannot impose any musical style upon the composer. Instead, we want a highly customizable environment. The system should offer a rich set of structures for composition and synthesis, combined with a high-level programming environment to express the musical ideas of the composer.

- Complex interactivity: The interactivity should not be reduced to the sequencing of pre-existing parts. Instead, it should be possible to generate the contents and structure of a piece on the fly according to a high-level description. In an interactive piece the user should be able to manipulate the structure of the piece using high-level tools. In addition, the use of events should not be limited to the setting of a control value. Indeed, events can provoke a complex response on the side of the system.

The environment we proposed in the text is entirely written in the Java programming language. In addition, it uses an embedded Scheme interpreter. This approach furnishes us with a dynamic programming environment that can be personalized and extended. The Scheme interpreter is written in Java, and the Scheme objects are implemented as Java objects. This allows a transparent use of Scheme primitives in the environment. In particular, we will use the Scheme procedures to describe the complex behavior of the system. In the text we called these functional objects *programs*. Programs are used in two basic elements of the environment:

- Events: they consist of an evaluation time, a program, and the arguments for the program. At the specified time, the program is evaluated with its arguments.

- Activities: they are the basic element of temporal composition and represent any activity in time. They consist of a location in time and a duration. How the activity is to be started is described by a start program; how it is to be stopped by a stop program.

In the text we also discussed the basic elements of sound synthesis:

- Synthesis processes: they are the basic signal processing unit.
- Control functions: they describe continuous varying parameter values in time. They describe, for example, the amplitude envelop of a sound. We defined control functions as continuous, multi-dimensional, reentrant time functions.

Synthesis processes can be created using a meta-class approach: *synthesis techniques* create new *synthesis voices*, synthesis voices create new synthesis processes. We have discussed additional structures to organize activities in a musical piece:

- Patterns: they are basic element of organization. A pattern groups a set of activities and the time relations between them. The pattern is also in charge of the reorganization of the activities when modifications are made. These changes may trickle down the musical structure and provoke an incremental update. This approach is inspired on the graphical layout managers found in the Java Abstract Window Toolkit. It can be extended to the technique of constraint propagation found in graphical interfaces.
- Motifs: they can be considered as a handle to a pattern. They allow a pattern to be inserted at several occasions in a music piece.

Additionally, *pattern modifiers* can be used to make variations of a pattern or to extend the organization of a pattern to other elements than time. When activities of unknown duration are used, causal relations can be used to express their temporal organization. These operators reorganize the start and stop programs of activities to assure a correct sequencing.

The environment also offers a rich, layered time model. This time model associates a local time axis to each level of the hierarchy in which a musical piece is organized. Additionally we have the following two objects:

- Time models: they map the time axis of one level of the hierarchy onto the axis of the next level. They can be used to deform time.

- Time context: they bundle all the time information of one activity in the composition. This information is used by the synthesis processes and control functions to map to global time of the synthesizer to the local time of the activity.

In the text we discussed the architecture of the system. The environment handles three tasks concurrently:

- Synthesis: A synthesizer object calls all the active synthesis processes for the sound synthesis. Sound can be output directly to the audio device.
- Event handling: an event thread handles all events that are sent to the system. These events can be sent from any source, including the synthesis processes.
- User interaction: an interface thread handles the interaction with the user thru a Scheme shell.

The presence of a real-time task (the synthesizer) and a garbage collector (the Scheme interpreter) in a multi-threaded environment poses a technical problem to guarantee hard real-time conditions. In the text we examined the possible interactions between the two tasks. We imposed the constraint that the synthesis processes cannot allocate new objects during the synthesis. Under these conditions the only synchronization between the two tasks is the root scanning (an atomic operation) and the write barrier (additional instructions for every pointer write operation). If the synthesis processes are developed using traditional real-time design techniques – they generally implement straightforward, easy to analyze signal processing techniques – the problem of the garbage collector reduces to the scheduling problem of concurrent tasks. Thus, real-time performance should be possible.

To use the environment in a distributed application, events can be send using the UDP/IP protocol. On the server side these events are handled by programs set by the user, similar to the handling of local events. Low-level, distributed events thus can use the full richness of the high-level programming environment. Interaction with the Scheme shell is realized over a TCP/IP connection. Real-time sound output over the network is not available as yet. For this we will move the sound output on top of the Real-Time Streaming Protocol (RTSP).

Future work

Although the environment, in its current state, represents a rich playground for music composition and sound synthesis, there are many issues that require additional study and development:

- Logical constraint propagation: Patterns organize activities according to a set of relations. These patterns verify the relations after each time manipulation. This approach directs us to the techniques of logical constraint propagation used in graphical interfaces. Further study should allow us to extend the structures for temporal composition and integrate the full power of constraint propagation techniques into the composition environment. Possible starting points are [BMSX97, FB93, FJLV98]

- Real-time behavior: The current prototype offers good time performances. For critical situations, however, better real-time guarantees are needed. This requires a more detailed study of the behavior of the system, in particular, the influence of the garbage collector. We have found it hard to come to a conclusion on the real-time capabilities of such a dynamic environment solely on the basis of the discussions found in the literature. A theoretical study should therefore be accompanied by a thorough evaluation of a carefully designed implementation of the garbage collector and the Java virtual machine.

- Formalization: We have presented a formal model of the environment. This model served as a guide-line for the implementation. Additional study must allow a more rigid formalization of the architecture of the system. We can refer to Hoare's communicating sequential processes, Petri-nets, Milner's π-calculus [Mil91], Pratt's work on the duality of time and information [Pra92], or scheduling algebras (see for example [vGR98]).

- Graphical editors: The current Scheme interface offers a very flexible programming environment. However, from experience we know that many composers are hesitant to pick up the Scheme programming. It will therefore be useful to develop graphical interfaces for particular programming tasks (the creation of synthesis patches, for example).

- Libraries: The current environment lacks a rich set of functions for algorithmic composition and modules for sound synthesis. In the future we hope to use the environment in artistic projects and collect the gained expertise in additional libraries.

Appendices

Appendix A

Reference of the Varèse-Scheme interface

This appendix can serve as a reference of the Varèse-Scheme interface. At start-up time, the Scheme interpreter loads the file ~/.varese.scm. This file in turn loads the functions definitions of the interface. The user can request to load additional initialization files.

The reference is divided into the following sections:

- Environment settings
- Synthesizer
- Programs, events, event scheduler, and event queue
- Activities
- Patterns, motifs, pattern modifiers, and time contexts
- Synthesis techniques and synthesis voices
- Scheme server and distributed event handling
- Control functions
- Synthesis processes
- Utility functions

Environment settings

The settings concern the parameters for the sound synthesis and output. They are generally set in the ˜/.varese.scm file. Once the environment is running they can be modified and inspected. To validate the modifications of the settings, they must be loaded by the synthesizer using the `load-settings` procedure.

`(set-sample-rate!` *v*`)`
Sets the sample rate of the synthesizer to *v*. The environment uses one sample rate for the sound input, the synthesis, and sound output.

`(set-sample-size!` *v*`)`
Sets the sample rate of the sound output to *v*. Currently supported sample sizes are 8 and 16 bits.

`(set-output-channels!` *v*`)`
Sets the number of output channels. The number of possible channels is determined by the audio device of the system.

`(set-input-channels!` *v*`)`
Sets the number of output channels. The number of possible channels is determined by the audio device of the system. Sound input is currently only supported on the SGI platform.

`(set-synth-buffer-size!` *v*`)`
Sets the size of the synthesis buffers. Sound synthesis will be done in in chunks of *v* samples at a time. *v* is best a power of two between 64 and 2048.

`(set-output-buffer-size!` *v*`)`
Sets the size of the buffer used internally by the audio device for the sound output. *v* is best a power of two between 512 and 8192.

`(set-input-buffer-size!` *v*`)`
Sets the size of the buffer used internally by the audio device for sound input. Currently not used.

`(set-output-type! `*s*`)`
Sets the type of sound output. *s* should be a string. The following output types are available:

- direct: Output to the audio device
- file: Output the a file
- socket: Output over a TCP/IP socket

Additional arguments for the output are set with the `set-output-arg!` procedure.

`(set-output-arg! `*s*`)`
Sets the additional arguments for the sound output. The arguments are passed as a string. The following arguments are required for the available output types:

- direct: None
- file: “file name”
- socket: “host-name:port”

`(get-sample-rate)`
Returns the current sample rate.

`(get-sample-size)`
Returns the current sample size.

`(get-output-channels)`
Returns the current number of output channels.

`(get-input-channels)`
Returns the current number of input channels.

`(get-synth-buffer-size)`
Returns the current size of the synthesis buffer.

`(get-output-buffer-size)`
Returns the current size of the audio device buffer for sound output.

`(get-input-buffer-size)`
Returns the current size of the audio device buffer for sound input.

Synthesizer

`(global-synthesizer)`
Returns a reference the global synthesizer object.

`(start)`
Sets the synthesizer to playing.

`(pause)`
Sets the synthesizer to paused. The clock is not reset, and the active synthesis processes are not killed. A subsequent call the `start` will continue the synthesis.

`(stop)`
Sets the synthesizer to stopped. The clock is reset to zero, and the active synthesis processes are killed.

`(get-media-unit)`
Returns the number of samples that have been synthesized. This number is not necessarily equal to the number of samples played due to internal buffering.

`(get-media-time)`
Returns the current media time in seconds. The current media time is calculated from the number of synthesized samples.

`(add` *id sp*`)`
Adds the synthesis process `sp` to the set of active synthesis processes. Give *sp* the identification number `id`.

`(kill` *id1 id2 ...*`)`

Kills the active synthesis processes with identification numbers *id1* , *id2*, ...

`(kill-all)`
Kills all active synthesis processes.

`(print)`
Prints the identification numbers and class type of all active synthesis processes.

`(load-settings)`
Loads the current settings. The sound output is closed and a new one is opened with the current settings. It is an error to call the procedure when the synthesizer is not stopped.

Programs, events, event scheduler, and event queue

`(program` *proc*`)`
Wraps the procedure *proc* in a new program object. Returns the new program object.

`(eval-program)`
Returns a new evaluation program.

`(new-event` *t prog args*`)`
Returns a new event object with evaluation time *t*, program *prog*, and arguments *args*. The *args* should be an object array (see `list->array`).

`(scheduler)`
Returns a new scheduler object. A scheduler is a synthesis process and can be added to the synthesizer.

`(scheduler-print` *sched*`)`
Prints out the current schedule of the scheduler.

`(scheduler-schedule` *sched evt*`)`
Schedules the event *evt* in the scheduler *sched*.

`(scheduler-erase` *sched*`)`
Erases all events of the scheduler *sched*.

`(global-scheduler)`
Returns the global scheduler. This scheduler is created systematically at the startup of the environment.

`(schedule` *e*`)`
Schedules the event *e* in the global scheduler. This is short for (scheduler-schedule (global-scheduler) e)

`(erase-schedule)`
Erases all events of the global scheduler. This is short for (scheduler-erase (global-scheduler))

`(event-queue` *size*`)`
Creates a new event queue with *size* entries.

`(event-queue-post` *q e*`)`
Posts the event *e* to the event queue *q*. If the event queue is filled, the event is not inserted. The procedure returns a boolean indicating whether the event is posted or not.

`(event-queue-get` *q*`)`
Returns the next event in the event queue *q*. When no events are available the calling thread blocks until an event is posted.

`(global-event-queue)`
Returns a reference to the global event queue. This event queue is created at startup time. Events posted to this queue will be handled by the environment's default event thread.

`(post` *e*`)`
Posts the event *e* to the global event queue. This is short for (event-queue-post (global-event-queue) e))

(`event` *t proc a1 a2 ...*)
The `event` procedure creates and schedules a new event with time *t*. The *proc* argument should be a Scheme procedure and is implicitly wrapped into a Varèse program. The arguments *a1*, *a2*, ... are stored in an object array. The newly created event object is then scheduled in the global scheduler. The procedure is a short notation for (post (new-event t (program proc) (array a1 ...)))

Activities

(`activity` *pos dur sprog sarg eprog earg*)
Returns a new activity with position *pos*, duration *dur*, start program *sprog*, start arguments *sarg*, stop program *eprog*, and stop arguments *earg*. The position can be any type, the duration should be a number. The arguments *sarg* and *earg* should be arrays (see `list->array`).

(`set-position!` *a pos*)
Sets the position of activity *a* to *pos*. This modification may induce a reorganization of the musical structure in which *a* is included. This reorganization may fail and result in an error. In that case the original position of *a* is retained and no changes are made to the original structure.

(`get-position` *a*)
Returns the position of activity *a*.

(`set-duration!` *a dur*)
Sets the duration of activity *a* to *dur*. This modification may induce a reorganization of the musical structure. If the reorganization fails, the original duration of *a* is kept and no changes are made to the structure.

(`get-duration` *a*)
Returns the duration of activity *a*.

(`stretch` *a f*)

Stretches the duration of activity *a* by a factor *f*. See also `set-duration!`. This is short for (set-duration! a (* (get-duration a) f))

`(get-start-prog` *a*`)`
Returns the start program of activity *a*.

`(get-start-arguments` *a*`)`
Returns the start arguments of activity *a* as an array.

`(get-stop-prog` *a*`)`
Returns the stop program of activity *a*.

`(get-stop-arguments` *a*`)`
Returns the stop arguments of activity *a* as an array.

`(set-start-prog!` *a prog*`)`
Sets the start program of activity *a* to *prog*.

`(set-start-arguments!` *a arg*`)`
Sets the start arguments of activity *a*. *arg* should be an array.

`(set-stop-prog!` *a prog*`)`
Sets the stop program of activity *a* to *prog*.

`(set-stop-arguments!` *a arg*`)`
Sets the stop arguments of activity *a*. *arg* should be an array.

`(get-pattern` *a*`)`
Returns a reference to the pattern in which activity *a* is included.

`(play` *a*`)`
Plays the activity *a*.

`(new-sound-activity` *pos dur voice num*`)`
Creates a new sound activity with position *pos*, duration *dur*, synthesis voice *voice*, and *num* number of voice arguments.

`(sound-activity` *pos dur voice freq amp a1 a2 ...*`)`

The procedure `sound-activity` creates and initializes a sound activity. The arguments *freq* and *amp* are control functions indicating the frequency and amplitude evolution of the sound activity. *a1*, *a2*, ... are additional arguments for the synthesis voice.

`(sound-activity-set-arg!` *a v i*`)`

Set the *i*-th argument of the synthesis voice of sound activity *a* to *v*.

`(set-freq!` *a freq*`)`

Set the frequency control function of sound activity *a* to *freq*.

`(set-amp!` *obj amp*`)`

Set the amplitude control function of sound activity *a* to *amp*.

`(note` *dur v mc db*`)`

Creates a new note object with duration *dur*, pitch *mc* (in midi-cents), and intensity *db* (in decibels). The note will be played using the synthesis voice *v*.

`(set-pitch!` *n mc*`)`

Set the pitch of the note *n* to *mc* (in midi-cents).

`(set-intensity!` *n db*`)`

Set the intensity of the note *n* to *db* (in decibels).

`(chord` *dur v mc-list db-list*`)`

Creates a new chord object with duration *dur*, pitches *mc-list* (in midi-cents), and intensities *db-list* (in decibels). The chord will be played using the synthesis voice *v*.

`(rest` *dur*`)`

Creates a new rest object of duration *dur*.

Patterns, motifs, pattern modifiers, and time contexts

`(pattern-add` *p a1 a2 ...*`)`
Adds the activities *a1*, *a2*, ... to the pattern *p*.

`(pattern-remove` *p a*`)`
Removes activity *a* from the pattern *p*.

`(pattern-remove-all` *p*`)`
Removes all activities from the pattern *p*.

`(pattern-duplicate` *p*`)`
Creates a new pattern, clones all the activities in pattern *p* and adds them to the new pattern.

`(set-time-model!` *p tm*`)`
Sets the time model of pattern *p* to *tm*.

`(get-time-model` *p*`)`
Returns the time model of pattern *p*.

`(add-modifier` *p m*`)`
Adds pattern modifier *m* to the pattern *p*.

`(remove-modifier` *p m*`)`
Removes pattern modifier *m* from the pattern *p*.

`(motif` *pos dur p*`)`
Creates a new motif with position *pos*, and duration *dur*. The motif obtains its contents from the pattern *p*.

`(motif-add` *m a1 a2 ...*`)`
Adds the activities *a1*, *a2*, ... to the pattern of motif *m*. It is short for (pattern-add (get-sub-pattern m) a1 a2 ...))

`(sound-motif` *pos dur p*`)`
Creates a new sound motif with position *pos*, and duration *dur*. The motif obtains its contents from the pattern *p*. The difference between sound motifs and regular motifs is that sound motifs have a frequency and amplitude parameter that can be used by the modifiers.

`(get-sub-pattern` *m*`)`
Returns a reference to the pattern that organizes the contents of motif *m*.

`(variation` *m*`)`
Creates a new motif with the same pattern as motif *m*. It is short for (motif 0 (get-duration m) (get-sub-pattern m))).

`(duplicate` *m*`)`
Creates a new motif with a duplicate of the pattern of motif *m*. It is short for (motif 0 (get-duration m) (pattern-duplicate (get-sub-pattern m))))

`(sequence-pattern)`
Creates a sequence pattern.

`(make-sequence` *dur*`)`
Creates a motif with duration *dur* with an empty sequence pattern. It is short for (motif 0 dur (sequence-pattern))).

`(sequence` *dur a1 a2 ...*`)`
Creates a motif with duration *dur* with a sequence pattern. The activities *a1*, *a2*, ... are added to the pattern.

`(parallel-pattern)`
Creates a parallel pattern.

`(make-parallel` *dur*`)`
Creates a motif with duration *dur* with an empty parallel pattern. It is short for (motif 0 dur (parallel-pattern))).

`(parallel` *our a1 a2 ...*`)`

Creates a motif with duration *dur* with a parallel pattern. The activities *a1*, *a2*, ... are added to the pattern.

`(portamento-modifier)`
Creates a new portamento modifier.

`(vibrato-modifier` *freq amp*`)`
Creates a new vibrato modifier.

`(context-get-start` *c*`)`
Returns the start of time context *c*.

`(context-get-duration` *c*`)`
Returns the duration of time context *c*.

Synthesis techniques and synthesis voices

`(synthesis-technique` *name descr prog num param*`)`
Defines a new synthesis technique with the name *name* and description *descr*. The synthesis processes will be created by the program *prog* that takes *num* arguments. The synthesis parameters *param* is a list of lists. There must be *num* sub-lists. Each sub-list describes one parameter. Its first element is the name, the second element the description, the third element the default value of the parameter.

`(synthesis-technique-print` *st*`)`
Prints out the name, description, and parameter information of the synthesis technique *st*.

`(synthesis-technique-set!` *st n p*`)`
Sets the parameter with the name *n* of synthesis technique *st* to the value *p*.

`(synthesis-voice` *st*`)`
Creates a synthesis voice with the synthesis technique *st*.

`(voice-print` *v*`)`
Prints out the name, description, and parameter information of the synthesis voice *v*.

`(voice-set!` *v n p*`)`
Sets the parameter with the name *n* of synthesis voice *v* to the value *p*.

`(voice-make` *v freq amp*`)`
Creates a synthesis process with the synthesis voice *v* with frequency control function *freq* and amplitude control function *amp*.

`(voice-make-list` *v a1 a2 ...*`)`
Creates a synthesis process with the synthesis voice *v* with the arguments *a1*, *a2*, ...

Scheme server and distributed event handling

`(server` *p*`)`
Starts the Scheme interpreter listening on a TCP/IP socket at port *p*.

`(udp-thread` *p*`)`
Starts a new thread to handle events send over UDP/IP to port *p*.

`(udp-thread-set!` *t p num*`)`
Sets program *p* to handle incoming UDP events with index number *num*. *t* is the event thread waiting for events.

Control functions

`(ctrl-dim` *c*`)`
Returns the dimension of control function *c*.

`(ctrl-value` *ctrl context t*`)`

Returns the value of control function *c* at time *t* in the time context *context*. The value #!null can be used for the time context.

`(new-const` *dim*`)`
Creates a new constant control function with dimension *dim*.

`(const` *v1 v2 ...*`)`
Creates and initializes a new constant control function with dimension *dim*. The values are set to *v1*, *v2*, ...

`(const-set!` *c v d*`)`
Sets the value of dimension *d* of the constant control function *c* to *v*.

`(const-get` *c d*`)`
Returns the value of dimension *d* of the constant control function *c*.

`(new-bpf` *dim f*`)`
Creates a new breakpoint function of dimension *dim* with *f* frames.

`(bpf` *dim v*`)`
Creates and initializes a new breakpoint function of dimension *dim*. The values *v* is a list of lists. Each sub-list specifies the values of one frame, and contains dim+1 values. The first element of a sub-list is the time of the frame, the remaining elements are the values of the breakpoints.

`(bpf-set-value!` *bpf v d f*`)`
Sets the value of the point at dimension *d* of frame *f* of the breakpoint function *bpf* to *v*.

`(bpf-get-value` *bpf d f*`)`
Returns the value of the point at dimension *d* of frame *f* of the breakpoint function *bpf*.

`(bpf-set-time!` *bpf t f*`)`
Sets the time of frame *f* of the breakpoint function *bpf* to *t*.

`(bpf-get-time` *bpf p*`)`

Returns the time of frame *f* of the breakpoint function *bpf*.

`(bpf-set-values!` *bpf v f*`)`

Sets all the values of frame *f* of breakpoint function *bpf*. The values *v* is a list. Its first element indicates the time of the frame, the remaining elements indicate the values of the points.

`(bpf-get-inv` *bpf*`)`

Returns the inverted breakpoint function of *bpf*. This procedure can be applied only to invertible, one dimensional breakpoint functions. `bpf-get-inv` can be used, for example, to invert time models.

`(ctrl-add` *c1 c2 ...*`)`

Returns a new control function as the sum of control functions *c1*, *c2*, ...

`(ctrl-sub` *c1 c2*`)`

Returns a new control function as the subtraction of control functions *c1* and *c2*.

`(ctrl-mul` *c1 c2 ...*`)`

Returns a new control function as the multiplication of control functions *c1*, *c2*, ...

`(ctrl-div` *c1 c2*`)`

Returns a new control function as the division of control functions *c1* and *c2*.

`(ctrl-neg` *c*`)`

Returns a new control function as the negation of control function *c*.

`(ctrl-inv` *c*`)`

Returns a new control function as the inverse of control function *c*.

`(ctrl-pow` *c1 c2*`)`

Returns a new control function as control function *c1* to the power of control function *c2*.

`(ctrl-mc-to-hz` *c*`)`
Returns a new control function that converts the value of control function *c* from midi-cents to Hertz.

`(ctrl-db-to-amp` *c*`)`
Returns a new control function that converts the value of control function *c* from decibels to linear amplitude.

`(ctrl-sin` *freq amp*`)`
Returns a new control function that outputs a sinusoidally varying value. The frequency of the sinusoid is a constant number value and indicated by *freq*. The amplitude is a control function given by *amp*.

`(time-level` *c lev*`)`
Returns a new control function that outputs the value of control function *c* after converting the local time to the time on level *lev* of the time context.

`(time-norm` *c*`)`
Returns a new control function that outputs the value of control function *c* after normalizing the local time to the duration of the time context.

Synthesis processes

`(set-context!` *sp c*`)`
Sets the time context of synthesis process *sp* to *c*.

`(get-id` *sp*`)`
Returns the identification number of the synthesis process *sp*.

`(reset` *sp*`)`
Resets the synthesis process *sp*. The synthesis process is as good as new and can be reused in the synthesis.

`(connect` *in n out*`)`

Connects synthesis process *out* to the *n*-th input of unit generator *in*. Note that *out* can be any synthesis process, not necessarily an unit generator.

(`sinewave` *freq amp*)
Creates a new sine-wave synthesis process with frequency control function *freq* and amplitude control function *amp*.

(`adsyn` *freq amp*)
Creates a new additive synthesis process with frequency control function *freq* and amplitude control function *amp*. Both *freq* and *amp* should be multi-dimensional.

(`sampler` *freq amp t*)
Creates a new sampler synthesis process with frequency control function *freq* and amplitude control function *amp*. The sampler will play thru the sample table *t* once and then output zero values.

(`oscil` *freq amp table*)
Creates a new oscillator synthesis process with frequency control function *freq* and amplitude control function *amp*. The oscillator will loop the sample table *t*.

(`pulse` *id freq amp*)
Creates a new pulse synthesis process with frequency control function *freq* and amplitude control function *amp*.

(`bandpass-filter` *fc q g*)
Creates a new bandpass-filter. The filter response is specified by three control functions: *fc* determines the central frequency, *q* the quality factor, and *g* the gain.

(`sound-table` *file f0*)
Creates a new sample table and loads the sample of file *file* into it. The currently supported file format is AIFF. *f0* indicates the base frequency of the sample stored in the table. It is used to determine the transposition factor in the synthesis.

(`sound-table-load` *t file*)
Loads the AIFF file *file* into the table *t*.

`(sin-table-init` *size*`)`
Creates a new table of size *size* and fills it with one period of a sine-wave.

`(sin-table)`
Short for (sin-table-init 8000)).

`(rect-table-init` *size ratio pos*`)`
Creates a new table of size *size* and fills it with one period of a rectangular wave. The value *ratio* (between 0 and 1) indicates the length of the first half of the wave compared to the length of the second half. If *pos* is #t the wave starts with the positive values; if *pos* is #f the wave starts with the negative values.

`(rect-table)`
Short for (rect-table-init 8000 0.5 #t))

`(saw-table-init` *size pos*`)`
Creates a new table of size *size* and fills it with one period of a rectangular wave. If *pos* id #t the values start at 1 and decrease to -1. If *pos* id #f the values grow from -1 to 1..

`(saw-table)`
Short for (saw-table-init 8000 #t))

`(harmonic-table-init` *dim n*`)`
Creates a new table of size *size* and fills it with the *n* sinusoidal, harmonic partials.

`(harmonic-table` *n*`)`
Short for (harmonic-table-init 8000 num)

Utility functions

`(random)`
Returns a random number between 0 and 1.

`(midi->hz` *m*`)`

Converts the midi key number *m* to a frequency value in Hertz

`(mc->hz` *mc*`)`

Converts *mc* from midi-cents to Hertz.

`(db->amp` *db*`)`

Converts *db* from decibels to amplitude.

`(new-array` *length*`)`

Creates a new object array of length *length*.

`(array-set!` *a e i*`)`

Sets element *i* of array *a* to *e*.

`(array-get` *a i*`)`

Returns the *i*-th element of array *a*.

`(array-length` *a*`)`

Returns the length of array *a*.

`(array` *e1 e2 ...*`)`

Creates a new array with elements *e1*, *e2*, ...

`(list->array` *l*`)`

Converts the list *l* to an array.

Bibliography

Bibliography

[AC95] G. Assayag and J.-P. Cholleton. Musique, nombres et ordinateurs. *La Recherche*, 26, Juillet-août 1995.

[AEL88] A.W. Appel, J. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design & Implementation*, pages 11–20. ACM, June 1988.

[Ago98] C.A. Agon. *OpenMusic : Un language visuel pour la composition assistée par ordinateur*. PhD thesis, Université Pierre et Marie Curie, Paris 6, Paris, France, 1998.

[AK89] D.P. Anderson and R. Kuivila. Continuous abstractions for discrete event languages. *Computer Music Journal*, 3(13):11–23, 1989.

[App87] A.W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[AR93] G. Assayag and C. Rueda. The music representation project at Ircam. In *Proceedings of the Int. Computer Music Conference*, Tokyo, Japan, 1993. Int. Computer Music Association.

[Ass96] G. Assayag. Openmusic. In *Proceedings of the Int. Computer Music Conference*, Hong Kong, 1996. Int. Computer Music Association.

[Bak78] H.G. Jr. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bal92] M. Balaban. *Music structures: Interleaving the temporal and hierarchical aspects in music*, pages 111–138. The AAAI Press/The MIT Press, 1992. ISBN 0-262-52170-9.

[Bar86] P Barbaud. *La musique discipline scientifique.* Dunod, Paris, 1986.

[BL89] H. Bestougeff and G Ligozat. *Outils logiques pour le traitement du temps.* Masson, Paris, 1989.

[BMSX97] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proc. of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, October 1997.

[Boe93] H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume SIGPLAN Notices 28, 6, pages 197–206, June 1993.

[Bot98] P. Bothner. Kawa: Compiling dynamic languages to the Java VM. In *Proceedings of the 1998 Usenix conference.* Usenix, 1998.

[BRP$^+$78] W. Buxton, W. Reeves, R. Patel, R. Baecker, and L. Mezei. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, 2(4), 1978. reproduced in *The Foundations of Computer Music*, Roads & Strawn, The MIT Press, 1985.

[BS95] A. Baird-Smith. *Distribution et Interpretation dans les Interfaces Homme-Machine.* PhD thesis, Universite Paris VI, Paris, France, 1995.

[CLF93] C. Cadoz, A Luciani, and J.-L. Florens. CORDIS-ANIMA: A modeling and simulation system for sound and image synthesis – the general formalism. *Computer Music Journal*, 17(1):19–29, Spring 1993.

[Cou92a] F. Courtot. CARLA: Knowledge acquisition and induction for computer assisted composition. *Interface*, 21:191–217, 1992.

[Cou92b] F. Courtot. *Logical representation and induction for computer assisted composition*, pages 157–181. The AAAI Press/The MIT Press, 1992. ISBN 0-262-52170-9.

[CSSL97] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence. Modelling applications for adaptive QoS-based resource management. In *Proceedings of the 2nd IEEE High Assurance Engineering Workshop*, August 1997.

[Dan89] R.B. Dannenberg. The Cannon score language. *Computer Music Journal*, 1(13):47–56, 1989.

[Dan93] R. Dannenberg. The implementation of Nyquist, a sound synthesis language. In Computer Music Association, editor, *Proceedings of the Int. Computer Music Conference*, pages 168–171, September 1993.

[Dan97] R.B. Dannenberg. Abstract time warping of compound events and signals. *Computer Music Journal*, 3(21):61–70, 1997.

[DDH97] R.B. Dannenberg, P. Desain, and H. Honing. *Programming language design for music*, pages 271–315. Swets and Zeitlinger, 1997. ISBN 90-265-1483-2.

[DH88] P. Desian and H. Honing. LOCO: A composition microworld in Logo. *Computer Music Journal*, 12(3), Fall 1988.

[DH92] P. Desain and H. Honing. Time functions function best as functions of multiple times. *Computer Music Journal*, 16(2):17–34, Summer 1992.

[DK94] A. Duda and C. Keramine. Structured temporal composition of multimedia data. In *Proceedings of the IEEE Int. Workshop on Multimedia Computing and Systems*, pages 140–151, Boston, May 1994. The Inst. of Electrical and Electronics Engineers.

[DP93] G. De Poli. Audio signal processing by computer. In G. Haus, editor, *Music Processing*. Oxford University Press, 1993. ISBN 0-19-816372-X.

[DR86] P. Dannenberg, R.B. McAvinney and D. Rubine. Artic: A functional language for real-time systems. *Computer Music Journal*, 4(10):67–78, 1986.

[EGA94] G. Eckel and R. González-Arroyo. Musically salient control abstractions for sound synthesis. In *Proceedings of the Int. Computer Music*

Conference, Aarhus, Danmark, 1994. Int. Computer Music Association.

[EIC95] G. Eckel, F. Iovino, and R. Caussé. Sound synthesis by physical modelling with Modalys. In *Proceedings of the Int. Symposium on Music Acoustics*, Le Normant, Dourdan, 1995.

[FB93] B. Freeman-Benson. Converting an existing user interface to use constraints. In *Proc. of the 1993 ACM Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.

[FJLV98] H. Fargier, M. Jourdan, N. Layaida, and Th. Vidal. Using temporal constraints networks to manage temporal scenario of multimedia documents. In *ECAI 98 workshop on Spatial and Temporal Reasoning*, Brighton, UK, 1998.

[GCFP97] Assayag G., Agon C., J. Fineberg, and Hanappe P. Object oriented visual environment for musical composition. In *Proceedings of the Int. Computer Music Conference*, San Francisco, 1997. Int. Computer Music Association.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steel. The Java language specification. Technical report, Sun Microsystems Computer Company, August 1996.

[GM95] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems Computer Company, October 1995.

[Hil70] L. Hiller. *Music composed with computers: A historical survey*, pages 42–96. Cornell Univ. Press, 1970.

[Hon93] H. Honing. Issues in the representation of time and structure in music. *Contemporary Music Review*, 9:221–239, 1993.

[Hon95] H. Honing. The vibrato problem: comparing two solutions. *Computer Music Journal*, 3(19):32–49, 1995.

[Jef92] K. Jeffay. On kernel support for real-time multimedia applications. In *Proc. Third IEEE Workshop on Workstation Operating Systems*, pages 39–46, Key Biscayne, FL, April 1992.

[JLR+98] M. Jourdan, N. Layaida, C. Roisin, L. Sabry-Ismael, and L. Tardif. Madeus, an authoring environment for interactive multimedia documents. September 1998.

[KCR98] R. Kelsey, W. Clinger, and J. (editors) Rees. Revised (5) report on the algorithmic language scheme. Technical report, February 1998.

[KD97] C. Keramine and A. Duda. Operator based composition of structured multimedia presentations. In *Proceedings of the 4th COST 237 Workshop – From Multimedia to Network Services*, Lisboa, December 1997.

[Kin82] C. Kingsley. Description of a very fast storage allocator. Technical report, Documentation of 4.2 BSD Unix malloc implementation, February 1982.

[LC94] O. Laumann and B. Carsten. Elk: The Extension Language Kit. *USENIX Computing Systems*, 7(4):419–449, 1994.

[Loe92] K. Loepere. Mach 3 kernel principles. Technical report, Open Software Foundation and Carnegie Mellon University, July 1992.

[Loy85] G. Loy. Musicians make a standard: The MIDI phenomenon. *Computer Music Journal*, 9(4):8–26, Winter 1985.

[Loy89] G. Loy. *Composing with computers: A survey of some compositional formalisms and music programming languages*, pages 291–396. MIT Press, 1989. ISBN 0-262-13241-9.

[MA93] J.D. Morrison and J.-M. Adrien. Mosaic: A framework for modal synthesis. *Computer Music Journal*, 17(1):45–56, Spring 1993.

[Mak75] J. Makhoul. Linear prediction: a tutorial review. In *Proceedings of the IEEE*, volume 63, pages 561–580. The Inst. of Electrical and Electronics Engineers, April 1975.

[Mat69] M.V. Mathews. *The technology of computer music.* The MIT Press, Cambridge, Massachussetts, 1969.

[Mil91] R. Milner. The polyadic πi-calculus: a tutorial, October 1991.

[MJ89] Laurson M. and Duthen J. Patchwork, a graphical language in preform. In *Proceedings of the Int. Computer Music Conference*, pages 172–175, San Franscisco, 1989. Int. Computer Music Association.

[MMR74] M.V. Mathew, F.R. Moore, and J.-C. Risset. Computers and future music. *Science*, 183:263–268, January 1974. Am. Ass. for the Advancement of Science.

[Moo78] J. Moorer. The use of the phase vocoder in computer music applications. *Journal of The Audio Engineering Society*, 26(1):42–45, 1978.

[NBPF96] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthread programming.* O'Reilly and Associates, Sebastopol, CA, 1996. ISBN 1-56592-115-1.

[NG95] K. Nilsen and H. Gao. The real-time behavior of dynamic memory management in C++, May 1995. IEEE Real-Time Technology and Applications Symposium.

[Nil] K. Nilsen. Embedded real-time development in the Java language.

[Nil94] K. Nilsen. Reliable real-time garbage collection of C++. *Computing Systems*, 7(4):467–504, Fall 1994.

[Obj96] Object Management Group. *The Common Object Request Broker: Architecture and specification*, 1996.

[OFL97] Y. Orlarey, D. Fober, and S. Letz. Elody: a Java+MidiShare based music composition environment. In *Proceedings of the Int. Computer Music Conference.* Int. Computer Music Association, 1997.

[Opp96] D. Oppenheim. DMIX: A multi faceted environment for composing and performing computer music. *Mathematics and Computers*, 1996.

[PB98] A. Petit-Bianco. Java garbage collection for real-time systems. *Dr. Dobb's Journal*, October 1998.

[PC98] F. Pachet and J. Carrive. *Recherches et applications en informatique musicale*, chapter Intervalles temporels circulaires et application à l'analyse harmonique. Hermès, Paris, France, 1998.

[POS96] POSIX. Portable operating system interface (posix) – part 1: System application: Program interface. Technical Report IEEE/ANSI Std 1003.1, IEEE. Information Technology, 1996.

[PPK99] D. Pressnitzer, R. D. Patterson, and K. Krumbholz. The lower limit of melodic pitch with filtered harmonic complexes. *Journal of the Acoustical Society of America*, 1999. Résumé, sous presse.

[Pra92] V.R. Pratt. The duality of time and information. In *Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR'92)*, volume 630 of *Lecture Notes in Computer Science*, pages 237–253. Springer-Verlag, August 1992.

[PSS$^+$98] D. Porcaro, N. Jaffe, P. Scandalis, J. Smith, , T. Stilson, and S. Van Duyne. SynthBuilder: A graphical rapid-prototyping tool for the development of music synthesis and effect patches on multiple platforms. *Computer Music Journal*, 2(22):35–44, Summer 1998.

[Puc91a] M. Puckette. Combining events and signal processing in the MAX programming environment. *Computer Music Journal*, 3(15):68–77, 1991.

[Puc91b] M. Puckette. FTS: a real-time monitor for multiprocessor music synthesis. *Computer Music Journal*, 3(15):58–67, 1991.

[RC84] X. Rodet and P. Cointe. FORMES: Composition and scheduling of processes. In C. Roads, editor, *The Music Machine*, Cambridge Massachusetts, 1984. MIT Press.

[Ris93] J.-C. Risset. Synthèse et matériau sonore. In *Les Cahiers de l'Ircam: La Synthèse Sonore*, Paris, France, 1993. Editions Ircam - Centre Georges-Pompidou.

[RL97] X. Rodet and A. Lefèvre. The Diphone program: New features, new synthesis methods and experience of musical use. In *Proceedings of the Int. Computer Music Conference*, Thessaloniki, Hellas, 1997.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[Roa96] C. Roads. *The computer music tutorial.* The MIT Press, Cambridge, Massachussetts, 1996.

[RPB93] X. Rodet, Y. Potard, and J.-B Barrière. The CHANT project: From the synthesis of the singing voice to the synthesis in general. In C. Roads, editor, *The Music Machine*, Cambridge Massachusetts, 1993. MIT Press.

[SD93] C. Semal and L. Demany. Further evidence for an autonomous processing of pitch in auditory short-term memory. *Journal of the Acoustical Society of America*, 94(3):1315–1322, 1993.

[SGH$^+$97] D. C. Schmidt, A. Gokhale, T.H. Harrison, D. Levine, and Cleeland Ch. TAO: A high-performance ORB endsystem architecture for real-time CORBA. RFI response to the OMG Special Interest Group on Real-time CORBA, 1997.

[SJ93] E. Saint-James. *La programmation applicative (de Lisp à la machine en passant par le lambda-calcul).* Hermès, Paris, France, 1993.

[Smi96] J.O. III Smith. Physical modeling synthesis update. *Computer Music Journal*, 20(2):44–56, Summer 1996.

[Smi97] J.O. III Smith. *Acoustical modeling using digital waveguides*, pages 221–263. Swets and Zeitlinger, 1997. ISBN 90-265-1483-2.

[SRL87] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical report, Department of Computer Science, CMU, 1987.

[SV95] D. C. Schmidt and S. Vinoski. Object interconnections: Introduction to distributed object computing. *C++ Report Magazine*, January 1995.

[Tau91] H. Taube. Common music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, 15(2), Summer 1991. MIT Press.

[TM97] M. Timmerman and J.-C. Monfret. Windows NT as a real-time OS? *Real-Time Magazine*, 2:6–14, 1997.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, October 1990.

[Var72] L. Varèse. *A Looking-Glass Diary.* Norton, New York, 1972.

[Ver86] B. Vercoe. *Csound: A Manual for the Audio Processing System and Supporting Programs with Tutorials.* Media Lab, MIT, 1986.

[vGR98] R. van Glabbeek and P. Rittgen. Scheduling algebra. Arbeitsberichte 12, Institut für Wirtschaftsinformatik, Universtität Koblenz-Landau, Germany, 1998. to appear as Report STAN-CS-TN-98-.., Dep. of Computer Science, Stanford University.

[Vin97] S. Vinoski. CORBA: Integrating diverse applications within distributed hetergeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[Wil94] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, 1994.

[WJ93] P. R. Wilson and M.S. Johnstone. Real-time non-copying garbage collection. In *ACM OOPSLA Workshop on Memory Management and Garbage Collection*, 1993.

[Xen90] Y. Xenakis. *Formalized music: Thoughts and mathematics in music.* Pendragon, 1990.

[Zor93] B. Zorn. The measured cost of conservative garbage collection. *Software – Practice and Experience*, 23(7):733–756, 1993.