Exploring the possibilities and limitations of Concurrent Programming for Multimedia Interaction and Visual Programming for Musical Constraint Satisfaction Problems

Mauricio Toro Bermúdez
mauriciotorob at gmail.com
Department of Computer Science
Pontificia Universidad Javeriana de Cali (PUJC)

Internship developed at:

Music Representation Team
Institut de recherche et coordination acoustique-musique (Ircam)

in collaboration with
AVISPA research group,
under the REACT project
funded by Colciencias, Colombia.

Supervised by:
Carlos Agón - Ircam
Gérard Assayag - Ircam - CNRS UMR 9912
Camilo Rueda - Pujc

December 15, 2008
Abstract

Multimedia interaction systems are inherently concurrent. Developing correct concurrent systems is difficult because we need to consider all the possible interactions between processes. To reason formally about concurrent systems, there are several concurrent process calculi. We developed multiple prototypes for real-time capable interpreters for both, Concurrent Constraint Programming (CCP) and Non-deterministic Timed Concurrent Constraint (ntcc). We found out that using lightweight threads to implement these interpreters is not appropriate for real-time (RT) interaction. Instead, we recommend using event-driven programming. Using this model of concurrency, we developed Ntccrt, an interpreter for ntcc capable of RT interaction. Ntccrt is based on encoding ntcc processes as Gecode propagators. Using Ntccrt, we executed some models in Pure Data. Due to our success using Gecode, we upgraded Gelisp, providing a graphical interface to solve musical Constraint Satisfaction Problems (CSP) in OpenMusic based on Gecode. In Gelisp, constraints, search heuristics, and optimization criteria can be represented graphically. Using Gelisp, we successfully solved a CSP proposed by compositor Michael Jarrell.

Keywords: concurrent constraint programming, constraint satisfaction problem, constraints, ntcc, gelisp, csp, interpreter, ccp, ntccrt, openmusic, real-time, gecol, gecode.
Acknowledgments

In no particular order, I would like thank to my parents for all the effort they put in my education since I was a child and their support during the development of my internship. I want to thank Camilo Rueda from Pujc and Gérard Assayag from Ircam, my two internship supervisors, for the valuable comments and their support. In addition, I would like to thank to Carlos Agón from Ircam, who was like a third supervisor for me. To Carlos Olarte from Lix, Jean Bresson from Ircam, Guido Tack from Saarland university, Christian Schulte from KTH, Luis Omar Quesada from 4C, Boris Mejias and Gustavo Gutierrez from UCL for their valuables comments and help while developing the real-time interpreter. Specially, to Gustavo for his enormous patience.

To my music teachers, who taught me all I know about music: Mauricio Santamaria, Alberto Riascos, and Juan Manuel Collazos. To my brother Carlos and my friend Juan Pablo García for their valuable comments and editing contributions to improve my reports and articles. To Antal Buss and Nestor Cataño for teaching me LaTex. To Mónica Posso from Pujc and Ghislaine Montagne from Ircam for their administrative work related with this work and my thesis. Martin Simmons and and Paul Graham for their help during the exploratory study about lightweight threads. To Killian Sprotte for all the help when I was upgrading Gecol.

For the modeling of our applications, I want to thank to Frank Valencia from Lix, Jorge Pérez from university of Bologna, and Arshia Cont from Ircam. In addition, I want to thank Fivos Maniatakos from Ircam for his valuable comments about the interpreter and the article about it, and Serge Lemouton for his valuable comments about Gelisp. In general, to all the people from Ircam and Avispa who helped me during my internship and finally, to all my friends and family who supported me during this work.
## Contents

1 Introduction ........................................... 4
   1.1 Lightweight threads for Common Lisp ............... 5
      1.1.1 Comparing threads by their context-switch duration .... 5
      1.1.2 Motivation: Developing real-time systems in Common Lisp ... 5
      1.1.3 Strategies to implement lightweight threads .......... 5
   1.2 Using ntcc for multimedia interaction ............... 5
      1.2.1 Motivation: A declarative approach for concurrency in data-flow languages 6
      1.2.2 Overview of other solutions ............................ 6
      1.2.3 Disadvantages of the other solutions .................. 6
      1.2.4 Our solution: Using ntcc to control concurrency in Max and Pd ... 6
   1.3 Specifying Constraint Satisfaction Problems (CSP) graphically .... 7
      1.3.1 CSP's for Music ...................................... 7
      1.3.2 Motivation: A graphical representation for CSP's ........ 7
      1.3.3 Our solution: Extending Gelisp ....................... 7
   1.4 Contributions ........................................ 8
      1.4.1 Gecol extension ...................................... 8
      1.4.2 Gelisp extension ...................................... 8
      1.4.3 Ntccrt .............................................. 8
   1.5 Organization ......................................... 8

2 Background ............................................ 9
   2.1 Implementation techniques for Lightweight threads ....... 9
      2.1.1 Event-driven programming .............................. 9
      2.1.2 Scheduler activations .................................. 10
      2.1.3 Co-routines ......................................... 10
      2.1.4 Stack-based threads ................................... 10
      2.1.5 Protothreads ......................................... 11
      2.1.6 Virtual machines with thread support ................. 11
   2.2 Concurrent Constraint Process (CCP) ................... 11
      2.2.1 Relation between propagators and CCP ................ 11
      2.2.2 Disadvantages ........................................ 12
   2.3 Non-deterministic Timed Concurrent Constraint (ntcc) .... 12
      2.3.1 Examples ............................................ 13
   2.4 Generic Constraint development Environment (Gecode) .... 15
      2.4.1 Constraints as Propagator Agents (CPA) ............... 15
      2.4.2 Advantages ......................................... 16

3 Developing real-time capable lightweight (lw) threads for Common Lisp ... 17
   3.1 Using continuations .................................... 17
      3.1.1 Motivation .......................................... 17
      3.1.2 Disadvantages ....................................... 17
      3.1.3 Tests .............................................. 17
   3.2 Using Event-Driven Programming .......................... 18
      3.2.1 Motivation .......................................... 18
4 Developing real-time capable interpreters for CCP

4.1 Our previous approaches
4.1.1 Gecode interfaces to Common Lisp
4.1.2 Threads in Lisp and C++
4.1.3 Event-driven programming in Lisp

4.2 Our solution: Encoding processes as Gecode propagators

4.3 Applications
4.3.1 Finding paths in a graph concurrently

5 Developing a real-time capable interpreter for ntcc

5.1 History of ntcc Interpreters
5.1.1 Lman
5.1.2 Ntccsim
5.1.3 Rueda’s interpreter

5.2 Our Solution: Encoding ntcc processes as propagators
5.2.1 Finite Domain (FD), Finite Set (FS), and Infinite rational trees
5.2.2 Representing the tell agent
5.2.3 Representing the when agent
5.2.4 Representing the Non-deterministic agent
5.2.5 Representing local variables
5.2.6 Representing timed processes
5.2.7 Representing the unless and persistent assignation agents
5.2.8 Representing ntcc definitions
5.2.9 Execution model

5.3 Applications
5.3.1 The dining philosophers
5.3.2 CCFOMI: Music Improvisation
5.3.3 Signal processing

6 Developing libraries to solve musical CSP’s in Common Lisp

6.1 Our previous approach: Extending Gecoll
6.2 Our solution: Extending Gelisp
6.2.1 Interface for Common Lisp
6.2.2 Graphical Interface for OpenMusic

6.3 Applications
6.3.1 All-interval series
6.3.2 Michael Jarrell’s CSP

7 Concluding Remarks

7.1 Results
7.2 Summary
7.3 Future Directions
7.3.1 Using a high-performance implementation of Common Lisp
7.3.2 Applications for the CCP interpreter
7.3.3 Using Gelisp for Ntccrt
7.3.4 Adding support for cells for Ntccrt
7.3.5 Developing an interpreter for pntcc
7.3.6 Developing an interpreter for rtcc
7.3.7 Adding other graphical interfaces for Ntccrt
7.3.8 Developing model checking tools for Ntccrt
7.3.9 Extending Rules2Cp for musical CSP’s in Gelisp
7.3.10 Adding more features to search in Gelisp
Chapter 1

Introduction

There is a dispute between computer scientists about the way to develop multimedia interaction systems. The first group argue that in order to implement real-time capable systems, those systems should be written directly in C++ for efficiency. The second group argue that those systems –inherently concurrent– should not be written directly in C++, because there is not a formalism to reason about synchronization and concurrency in C++. The second group propose modeling those systems using a formalism with formal semantics and verification procedures, and execute those models on a real-time capable interpreter.

Although several formalisms to model concurrent systems have been developed in the last two decades, we argue that there is not an efficient and generic interpreter to run multimedia interaction systems in real-time. This problem led us to a research about developing a real-time capable interpreter for a well-know formalism: Non-deterministic Timed Concurrent Constraint (ntcc) [25].

In order to develop the interpreter, we developed prototypes for a generic implementation of lightweight (lw) threads for Common Lisp. Since many applications for computer music are written in Common Lisp, we though that creating an efficient implementation of lw threads for this language would allow us to write real-time capable multimedia interaction systems as well as interpreters for Concurrent Constraint Programming (CCP) [11] (the ancestor of ntcc). Then extend the CCP interpreter for ntcc. The reader should be aware that most Common Lisp implementation does not provide lw threads.

We found out that our implementations of lw threads are efficient to model a variety of concurrent systems in Common Lisp, but they are not efficient for CCP and ntcc interpreters. However, we also found out that event-driven programming suits very good the concurrency control of a real-time capable CCP interpreter. That way we developed a CCP interpreter. Then, we extended the CCP interpreter to support timed processes and non-deterministic choice, creating a real-time capable interpreter for ntcc.

Our ntcc interpreter is called Ntcrt. Using Ntcrt we executed –in real-time– ntcc specifications of multimedia interaction systems. As far as we know, this is the only software providing a generic framework to specify and execute in real-time multimedia interaction systems modeled with ntcc.

Since the ntcc interpreter is based on the constraint solving library Gecode [16] and the results were outstanding, we upgraded two Common Lisp wrappers for Gecode, Gecol and Gelisp. Gecol and Gelisp were originally developed for Gecode 1.3.1 and current version of Gecode es 2.2.0. After upgrading them to current version of Gecode, we developed a graphical interface for Openmusic (OM) [5] for Gelisp. The goal was to use Gecode to solve Constraint Satisfaction Problems (CSP) for computer music and also using the interface (in the future) to execute ntcc specifications in Common Lisp.

The rest of this introduction is organized as follows. Section 1.1 introduces the concept of lightweight threads and the different strategies to implement them. Section 1.2 explains how we can use Ntcrt to develop multimedia interaction systems and execute them in Pure Data (Pd) [31] and Max/Msp [32]. Section 1.3 gives the motivation of solving CSP’s graphically and extending Gelisp for that purpose. Finally, Section 1.4 presents the software we developed as a contribution
1.1 Lightweight threads for Common Lisp

In computer science, a *continuation* is an abstraction of the processor registers, the *events* are an abstraction of the hardware interruptions and a *thread* represents a sequential flow control or an abstraction of a processor.

Sometimes, *threads* are described by their weight, meaning how much contextual information must be saved for a given thread in order to schedule them \[58\]. For instance, the context of a Unix process includes the hardware register, the kernel stack, user-level stack, process id, and so on. The time required to switch from one Unix process to another is large (thousands of microseconds), for that reason those are called *heavyweight threads*.

Modern operating systems kernels, such as Mac OS X and Mach, allow to have multiple threads in the same process, decreasing the amount of context that must be saved with each one. These *threads* are called *medium-weight threads* and it takes hundreds of microseconds to switch between them \[51\].

1.1.1 Comparing threads by their context-switch duration

When all context and *thread* operations are exposed at user-level, each application needs only a minimal amount of context information saved with it, so that context switching can be reduced to tens of microseconds. These are called *lightweight (lw) threads*. For instance, *lightweight threads* used by the Java VM outperform *linux threads* on *thread* activation and synchronization because *thread* management operations do not need to cross kernel protection boundaries. But, *linux native threads* have better performance on I/O operations \[50\]. Additionally, since *lightweight threads* may block all the other *threads* when performing a blocking I/O operation, it is necessary to use asynchronous I/O operations, adding complexity and increasing the latency for I/O operations.

1.1.2 Motivation: Developing real-time systems in Common Lisp

Most Common Lisp implementations such as Lispworks, SBCL, and MCL provide medium-weight threads (usually called Lisp processes). They are usually based on *pthreads* (a portable implementation of medium-weight threads in C).

Those threads have two limitations, one is the amount of threads that can be working at the same time. Usually, we can have hundreds of threads, opposed to lw threads where we can have thousands of them. The other problem is the context-switch time. Medium-weight threads are significantly slower than lw threads, being incompatible with real-time interaction. To solve that problem, we explored different strategies to implement lw threads in Common Lisp.

1.1.3 Strategies to implement lightweight threads

Strategies to implement *Lightweight threads* include, but are not limited to: Scheduler activations \[3\], a threading mechanism that maps \( n \) user level *threads* into some \( m \) kernel *threads*; Prothethreads \[11\], an abstraction that reduces the complexity of *Event-based programs*; virtual machine with thread support \[7\], supporting the concurrent execution of multiple *threads* in the traditional way; Coroutines \[20\], allowing multiple entry points, suspending and resuming execution at certain locations; Continuations \[8\, 17\], an abstraction of the processor registers commonly used in functional languages; multiple stack based *threads* \[35\], where we have an scheduler in charge of providing a fair execution to all *threads*; and *Event-driven programming* \[13\], where a *dispatcher* is in charge of processing events (stored in a queue) according to *event handler* for each type of event.

1.2 Using ntcc for multimedia interaction

We propose a new way to synchronize concurrent processes in signal processing languages such as Pure Data and Max. Although complex concurrent processes can be programmed extending
these languages with C++, we argue that writing those externals is difficult and time-demanding. We propose using the ntcc formalism to specify concurrent processes and execute multimedia interaction systems in real-time with our tool, Ntccrt.

1.2.1 Motivation: A declarative approach for concurrency in data-flow languages

During the last two decades, several graphical data-flow programming languages have been developed for signal processing and composition of interactive computer music. The idea behind these programming languages is controlling messages, audio and video signals by connecting graphical objects via inlets and outlets.

The graphic environment facilitates the programming of interactive multimedia applications for non-computer scientists according to the principles of Human-computer interaction. Examples of these programming languages are Pure Data (Pd [31]) and Max [32] developed by Miller Puckette.

A computation in a data flow program starts by receiving an input. After that, the input goes through multiple graphical objects. Each of them transforms the input into new messages, audio, or video signals. These graphical objects can receive an input and transform it at any time. For that reason, data-flow languages are inherently concurrent. The problem is that synchronizing processes depending on multiple conditions is not trivial and may require sending and receiving multiple complex messages (using complex data structures).

1.2.2 Overview of other solutions

To program complex concurrent applications in Max and Pd usually we need to extend them by creating externals (i.e., binary plugins) in C++, Python, Ruby, Scheme, or other programming languages.

Another approach is using the Flext library. Flext provides a unique interface to write externals for both, Pd and Max in the C++ language. It also provides an interface to write threads in different threading systems available for the C++ programming language.

Finally, there is an approach used by the improvisation software Omax [4]. This software is composed by two modules. One module is in charge of signal processing (written in Max) and the other one is in charge of concurrency control and style learning (written in OpenMusic). The concurrency control is made using Lisp processes (medium-weight threads found in many Common Lisp implementations) and share-state concurrency. Unfortunately, all of them require writing complex concurrent processes in C++ or Lisp.

1.2.3 Disadvantages of the other solutions

Synchronization provided by most programming languages –such as C++ and Lisp– is made by using locks, semaphores, monitors, or other shared-state concurrency abstractions. Writing correct programs using that model is difficult because it is required to specify the locks for variables, threads, shared-memory areas, etc.

1.2.4 Our solution: Using ntcc to control concurrency in Max and Pd

In this work, we propose using ntcc to manage concurrency in data-flow programs. Ntcc is a formalism where we can model reactive systems with synchronous, asynchronous and/or nondeterministic processes. Additionally, it provides multiple agents who can reason about partial information represented by constraints. The ntcc formalism and extensions of it have been used to model interactive systems such as: an audio processing framework [38], musical improvisation systems [36], [29], [42], and interactive scores [2], [42]. The novelty of this approach is specifying concurrency in declarative way.

This solution would be incomplete if the designer of the system would have to write an efficient implementation –in a programming language– of every system he designs. Fortunately, after modelling and proving properties of systems modeled in ntcc, it is possible to run those models by using interpreters. In fact, there are three interpreters available: Lman [24], ntccSim
(http://avispa.puj.edu.co) and Rueda’s interpreter, unfortunately none of them are able to achieve real-time multimedia interaction. Real-time interaction means a response time fast enough to interact with human players while they do not observe a delay in the communication.

In order to fix that inconvenient, we built an interpreter in the C++ programming language, capable of real-time (Ntccrt). Ntccrt uses the Generic Constraints Development Environment (Gecode) to manage constraints and concurrency, and Flexi for portability. Additionally we provide an interface to the music composition environment OpenMusic, allowing the user to specify (graphically) ntcc specifications and translating them to either stand-alone programs interacting with the real-time library Midishare, or externals for Pd and Max.

1.3 Specifying Constraint Satisfaction Problems (CSP) graphically

A Constraint Satisfaction Problem (CSP) is a mathematical problem where one must find objects that satisfy a number of constraints (i.e., criteria over those variables). We extended Gelisp, a library to represent musical CSP’s and search heuristics graphically. We provide two versions, one for Common Lisp and one for OpenMusic. Using this library, we modeled a CSP proposed by compositor Michael Jarrell and solved it successfully.

1.3.1 CSP’s for Music

CSP’s provide a declarative way to represent combinatorial problems, specifying constraints instead of a sequence of steps to find the solution (as used in imperative programming). Additionally, it is possible to specify rules to choose between branches during search (i.e., heuristics).

CSP’s in music are used to solve harmonic, rhythmic or melodic problems. In addition, they can be used for automatic generation of musical structures satisfying a set of rules. For instance, the classical All-interval series, where we need to find 12 different notes with different intervals.

In order to solve a CSP we have two approaches. One, is using a Constraint Programming language such as Prolog or Mozart-Oz, and the other one is using constraint solving tool-kits usually written in C++, but attachable to traditional programming languages such as Common Lisp.

1.3.2 Motivation: A graphical representation for CSP’s

Using Constraint Programming languages or constraint solving tool-kits to solve CSP’s is difficult because they usually require deep knowledge on C++ or logic programming. For that reason, several graphical constraint solving libraries for OpenMusic (OM) have been developed in the last decade.

Currently, there are four tools to solve CSP’s in OpenMusic. OmSituation generates music based on constraints, OmRe finds structures corresponding to rhythmical constraints, OmClouds finds approximated solutions to a CSP, and OMBacktrack is a wrapper for the constraint solving library Screamer.

A good graphical constraint solving library to solve musical CSP’s should provide graphical representations to choose heuristics for the search, post multiple kind of constraints graphically without using loops and recursion, and perform search and propagation using state-of-art algorithms.

Unfortunately, OmRC and OmSituation are designed to solve specific problems. OmBacktrack is no longer available for current versions of OM. Finally, OmClouds does not always provide a solution satisfying all the constraints, which is necessary for many musical problems.

1.3.3 Our solution: Extending Gelisp

Gelisp is a library to solve CSP’s Common Lisp. Gelisp is a wrapper for the constraint solving library Gecode. It was originally developed by Rueda in 2006 and we modified it to work with
current versions of OM and Gecode. Furthermore, we added support to model CSP’s and search heuristics graphically. The novelty of Gelisp is to provide an efficient graphical representation for search heuristics, optimization criteria, and high-level constraints such as “all the intervals of a sequence must be different”.

Gecode works on different operative systems and is currently being used as the constraint library for Alice[33] and Mozart-Oz, therefore it is very likely to be maintained for a long time. Furthermore, it provides an extensible API, allowing us to create new propagators and user-defined search engines. For instance, we can extend Gecode to reason about trees and graphs. Finally, Gecode’s performance is better than the constraints solving tool-kits used in Sicstus Prolog and Mozart-Oz (according to the benchmarks presented in http://www.gecode.org).

1.4 Contributions

1.4.1 Gecol extension

Our first approach to provide an interface for Gecode to Common Lisp was extending Gecol to work with current version of Gecode. Examples, sources, and binaries can be found at http://common-lisp.net/project/gecol/

1.4.2 Gelisp extension

An extension to Gelisp to work with current version of Gecode. We also provide a graphical interface for OpenMusic. Examples, sources, and binaries can be found at http://gelisp.sourceforge.net. An article about Gelisp is to be published next year [52].

1.4.3 Ntccrt

A real-time capable interpreter for ntcc. Examples, sources and binaries can be found at http://ntccrt.sourceforge.net. An article about Ntccrt is to be published next year [53].

1.5 Organization

In what follows, we describe the structure of this report. Chapter 2 presents the background describing briefly the strategies to implement lightweight threads, CCP, ntcc, Gecode. Chapter 3 presents and evaluates two alternatives to implement lightweight threads for Common Lisp, presenting results and tests for each one. Chapter 4 explains different strategies to implement a real-time capable interpreter for CCP and an application. Chapter 5 explains the design and implementation of Ntccrt and gives three applications in the computer music domain. Chapter 6 explains the design and implementation of Gelisp and presents two applications solved graphically. Chapter 7 gives some concluding remarks and explains future work.
Chapter 2

Background

2.1 Implementation techniques for Lightweight threads

In this chapter, we explore different strategies to implement lightweight threads. We also explain the advantages and disadvantages of each of them.

2.1.1 Event-driven programming

*Event-based programs* are typically driven by a loop that polls for events and executes the appropriate callback when the event occurs. This means that the flow of the program is determined by sensor outputs, user actions, or messages from other programs.

They tend to have more stable performance under heavy load than threaded programs according to [10]. However, when they are used with synchronous I/O operations, it is necessary to rewrite the program to use asynchronous I/O.

On the other hand, Ron von Behren et al [57] argue that although *Event-based programs* have been used to obtain good performance in high concurrency systems, if there is a good implementation of *threads* with a tight integration with the compiler, it is possible to obtain similar or higher performance with *threads* than with *events*.

In order to implement this model it is required: a *dispatcher*, which takes the events and call the appropriate handler; an *event queue*, which stores the events when the dispatcher is busy; and different *handlers* for each type of events [13]. The diagram of figure 2.1 represents this model.

![Figure 2.1: Event Driven Programming Control Flow](image)

When the events can change the “state” of the program, it is necessary to have a *Finite State Machine (FSM)* to keep track of the current state of the program [13].
2.1.2 Scheduler activations

Scheduler activations \[3\] is a threading mechanism that maps N user level threads into some M kernel threads. This takes the advantages from the kernel-level (“1:1”) and the user-level (“N:1”) threading. Scheduler activations for Linux OS were implemented in two modules: a patch for the linux kernel and the user-level part was developed by the Informatics Research Laboratory of Bordeaux (LABRI) in a library called Marcel threads \[6\]. The disadvantage is that they are not OS portable.

2.1.3 Co-routines

A co-routine is a non-preemptive thread. They generalize subroutines to allow multiple entry points, suspending and resuming execution at certain locations \[20\].

In the C language there is a library called the Portable Coroutine Library (libpcl). High-order programming languages such as Python and Ruby have support for them also. Scheme coroutines are made using continuations, which are a functional object containing the state of the computation. When the co-routine is evaluated, the store computation is restarted where it left off \[16\].

2.1.4 Stack-based threads

Each thread is represented by using an structure that contains thread ID, execution context, priority and the thread stack. Additionally, there is an scheduler to allow the concurrent execution of all threads \[49\].

Thread scheduling

The scheduler is in charge of providing a fair execution to all threads, a fair execution means that every thread will eventually execute. There is a runnable pool containing the runnable threads, when a new thread is created it is added to the runnable pool. There is also a suspended pool, where the threads that are “waiting” remain until their waiting condition becomes true. Finally, there are a terminated pool and the current thread \[44\], \[56\]. Figure 2.2 explains the thread states and their transitions.

![Figure 2.2: Thread states and their transitions \[44\]](image)

There are many scheduling policies such as Priorities, First Come First Served, Shortest Process Next, Shortest Remaining Time, and Round Robin (RR) \[30\]. We will focus in RR, which makes a fair execution to all threads and it is commonly used to model interactive systems. RR keeps all the threads in a queue and it guarantees that processor time is put equitably over the threads.

The time slice given to each thread cannot be too small because it will cause an overhead of queue management, however it cannot be too large because it will not be useful for interactive systems. There is another approach, the counting approach; which counts computation steps and give the same number of them to each thread, it is often used in real-time systems \[56\].

Priorities are important when we need to give more processor time to some threads than others. But, high priority threads should not starve low priority threads. When we want to mix priorities and Round Robin, there are different possible approaches: processes on an equal priority are
addressed in a round-robin manner; the time slice duration can change according to the priority; and the one used by Mozart Oz, where every tenth time slice of a high priority thread, a medium priority thread is give one slice and similarly with the medium and low priority threads [50].

When a thread creates a child thread, the child is given the same priority as the parent to avoid timing bugs such as the priority inversion, which is an scenario where a low priority task holds a shared resource that is required by a high priority task, causing the delayed execution of the high priority task.

Synchronization

Multiple solutions have been proposed for synchronization, such as: semaphores, monitors, message-passing, locks, etc. All of them accesses to a shared state, making very difficult to write correct concurrent programs. Another approach is declarative concurrency [55].

It consists of several threads of statements executing concurrently and communicating via a shared store, the store only contains logic variables. Those variables have two states: bound and not bound. When a variable is required for a computation and it has not been bound, the thread yields until the variable is bound, this way the synchronization is achieved.

2.1.5 Protothreads

Protothreads [11] propose an abstraction that reduces the complexity of Event-based programs. Using protothreads, it is possible to perform conditional blocking on top of Event-based programs, without the overhead of multi-threading which includes multiple stacks. They use local continuations, which behaves like continuations, but they do not save the stack.

2.1.6 Virtual machines with thread support

There are virtual machines with thread support such as the Simple Extensible Abstract Machine (SEAM) [7]. SEAM has been used to implement a naive Java Virtual Machine and the Alice language [21]. SEAM supports the concurrent execution of multiple threads in the traditional way: there is a stack of activation records, each record correspond to task to be executed. The scheduler coordinates the execution of multiple threads and the preemption. There are too disadvantages with SEAM: When a thread is blocked in a function outside SEAM, all the other threads get blocked too, and it does not provide good abstractions for implementing a system that needs to be concurrent with respect to external processes.

2.2 Concurrent Constraint Process (CCP)

Concurrent Constraint Programming (CCP) is as a model for concurrent systems. In CCP, a concurrent system is modeled in terms of constraints over the system variables and in terms of agents interacting with partial information obtained from those variables. A constraint is a formula representing partial information about the values of some of the system variables.

2.2.1 Relation between propagators and CCP

Programming languages based on the CCP model, provide a propagator for each constraint. Propagators can be seen as operators reducing the set of possible values for some variables. For instance, in a system with variables pitch1 and pitch2 taking MIDI values (each MIDI pitch unit represents a semi-tone), the constraint pitch1 > pitch2 + 2 specifies possible values for pitch1 and pitch2 (where pitch1 is at least one tone higher than pitch2). The CCP model includes a set of constraints and a entailment relation |= between constraints. This relation gives a way of deducing a constraint from the information supplied by other constraints.

The idea of the CCP model is to accumulate information in a store. This information is represented by constraints. The information on the store can increase but it cannot decrease. Concurrent processes interact with the store by either adding more information or by asking if some constraint can be deduced from the current store. If the constraint cannot be deduced,
this process blocks until there is enough information to deduce the constraint \[36\]. Consider for example, 4 agents interacting concurrently (fig. 2.3). The processes `tell (pitch_1 > pitch_2 + 2)` and `tell (pitch_2 > 60)` add new information to the `store`. The processes `ask (pitch_1 > 58) \rightarrow P` and `ask (pitch_1 = 58) \rightarrow Q` launch process `P` and `Q` respectively, when their condition can be entailed from the `store`. The reader may notice that process `ask (pitch_1 > 58) \rightarrow P` launches process `P`, but the process `ask (pitch_1 = 58) \rightarrow Q` will be suspended until its condition can be entailed from the `store`. (fig. 2.4).

Formally, the CCP model is based on the idea of a constraint system. “A constraint system is a structure \(<D, \vdash, \text{Var}>\) where \(D\) is a (countable) set of primitive constraints (or tokens), \(\vdash \in D \times D\) is an inference relation (logical entailment) that relates tokens to tokens and \(\text{Var}\) is an infinite set of variables” [41]. A (non primitive) constraint can be composed out of primitive constraints.

According to Rueda, the formal definition of CCP does not specify which types of constraints can be used. Thus, a constraint system can be adapted to many needs depending on the set \(D\). For instance, finite domain (Fd) constrains provides expressions such as \(x \in R\), where \(R\) is a set of ranges of integers. Constraints systems may also include expressions over trees, graphs, sets, etc [35].

\[\text{STORE}\]

\begin{verbatim}
tell pitch2 > 60
ask pitch1 > 58 do P

tell pitch1 > pitch2 + 2
ask pitch1 = 58 do Q
\end{verbatim}

Figure 2.3: Process interaction in CCP

\[\text{P}\]

\[\text{STORE}\]

\begin{verbatim}
pitch2 > 60
pitch1 > pitch2 + 2
ask pitch1 = 58 do Q
\end{verbatim}

Figure 2.4: Process interaction in CCP (2)

### 2.2.2 Disadvantages

Valencia and Rueda argue that the CCP model posses difficulties for modeling reactive systems where information on a given variable changes depending on the interactions of a system with its environment. The problem arises because information can only be added to the `store`, not deleted nor changed [39]. Since a machine improvisation system is a reactive system, we need to explore extensions of CCP to model this system in an easy and natural way.

### 2.3 Non-deterministic Timed Concurrent Constraint (ntcc)

\textbf{ntcc} introduces the notion of discrete time as a sequence of time-units. Each \textit{time-unit} starts with an empty `store` and it adds to the `store` the information received from the environment (i.e., the input received each \textit{time-unit}), then it executes all the processes corresponding to that \textit{time-unit}.
Opposed to the CCP model, in \texttt{ntcc} we can model variables changing through time, because they can change values from a time-unit to another.

2.3.1 Examples

The tell agent

Following, we give some examples of how the computational agents of \texttt{ntcc} can be used. Further formal definitions can be found in \cite{25} and a summary can be found in table \ref{table:ntcc_agents}. Using the \texttt{tell} agent is possible to add constraints such as \texttt{tell}(pitch$_1$ = 60) (meaning the pitch$_1$ must be equal to 60) or \texttt{tell}(60 < pitch$_2$ < 100) (meaning that pitch$_2$ is an integer between 60 and 100).

The when agent

The \texttt{when} agent can be used to describe how the system reacts to different events, for instance \texttt{when} (pitch$_1$ = 48 \land pitch$_2$ = 52 \land pitch$_3$ = 55) \texttt{do tell}(CMayor = true) is a process reacting as soon as the pitch sequence C, E, G (represented as 48, 52, 55 in MIDI notation) has been played, adding the constraint CMayor = true to the store in the current time-unit.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{tell} (c)</td>
<td>Adds the constraint c to the current store</td>
</tr>
<tr>
<td>\texttt{when} (c) \texttt{do} A</td>
<td>if c holds now run A</td>
</tr>
<tr>
<td>\texttt{local} (x) \texttt{in} P</td>
<td>runs P with local variable x</td>
</tr>
<tr>
<td>A $\parallel$ B</td>
<td>Parallel composition</td>
</tr>
<tr>
<td>\texttt{next} A</td>
<td>Runs A at the next time-unit</td>
</tr>
<tr>
<td>unless (c) \texttt{next} A</td>
<td>unless c can be inferred now, run A</td>
</tr>
<tr>
<td>$\sum_{i \in I}$ \texttt{when} (c$_i$) \texttt{do} P$_i$</td>
<td>Non deterministically chooses P$_i$ s.t. (c$_i$) holds</td>
</tr>
<tr>
<td>*P</td>
<td>Delays P indefinitely (not forever)</td>
</tr>
<tr>
<td>!P</td>
<td>Executes P each time-unit (from now)</td>
</tr>
</tbody>
</table>

Table 2.1: \texttt{ntcc} Agents

The parallel agent

Parallel composition ($\parallel$) allows us to represent concurrent processes, for instance \texttt{tell} (pitch$_1$ = 62) $\parallel$ \texttt{when} 48 < pitch$_1$ < 59 \texttt{do tell}(Instrument = 1) is a process telling the store that pitch$_1$ is 62 and concurrently reacts when pitch$_1$ is in the octave -1, assigning instrument to 1. The number one represents the acoustic piano in MIDI notation.

The next agent

The \texttt{next} agent is useful when we want to model variables changing through time, for instance \texttt{when} (pitch$_1$ = 60) \texttt{next tell} (pitch$_1$ <> 60), means that if pitch$_1$ is equal to 60 in the current time-unit, it will be different from 60 in the next time-unit (see figure \ref{fig:next_agent}).

The unless agent

The \texttt{unless} agent is useful to model systems reacting when a condition is not satisfied or it cannot be deduced from the store. For instance, \texttt{unless} (pitch$_1$ = 60) \texttt{next tell} (lastpitch <> 60), reacts when pitch$_1$ is different from 60 or it cannot be deduced from the store (i.e., pitch$_1$ was not played in the current time-unit), telling the store in the next time-unit that lastpitch is not 60 (fig. \ref{fig:unless_agent}).
The star agent
The star (*) agent can be used to delay the end of a music process indefinitely, but not forever. For instance, \texttt{*tell} (End = true). The ! agent executes a certain process each time-unit. For instance, \texttt{!tell} (PlaySong = true). The $\sum$ agent is used to model non-deterministic choices. For instance, $\sum_{i\in\{48,52,55\}}$ \texttt{when true do} \texttt{tell} (pitch = i) models a system where each time-unit, a note is chosen from the C major chord (C,E or G) to be played (fig. 2.7). $\sum_{i\in\{48,52,55\}}$ \texttt{when true do} \texttt{tell} (pitch = i) can be expressed as \texttt{tell} (pitch = 48) + \texttt{tell} (pitch = 52) + \texttt{tell} (pitch = 55).

Derived agents
The agents presented in table 2.2 are derived from the basic operators. The agent $A + B$ non-deterministically chooses to execute either $A$ or $B$. The persistent assignation process $x \leftarrow t$ change the value of $x$ to the current value of $t$ in the following time units. In a similar way, the agents in table 2.3 are used to model cells. Cells are variables which value can be re-assigned in terms of its previous value. For instance, $x : (z)$ creates a new cell $x$ with initial value $z$, $x \leftarrow g(x)$ change the value of a cell, and $exch_g[x,y]$ exchanges the value of cell $x$ and $z$. The reader may notice that using cells is different from $x \leftarrow t$ which changes the value of $x$ only once.
Figure 2.7: Example of the execution of a non-deterministic agent in ntc.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B \sum_{i \in {1,2}} \text{when } i = 1 \text{ do } A</td>
<td>\text{when } i = 2 \text{ do } B$</td>
</tr>
<tr>
<td>$x \leftarrow t$ local $v$ in $\sum_{v} \text{when } t = v \text{ do } \text{next } !\text{tell } (x = v)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Derived ntc Agents

<table>
<thead>
<tr>
<th>Agent</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : (z)$</td>
<td>tell$(x = z)</td>
</tr>
<tr>
<td>$x : \leftarrow g(x)$</td>
<td>local $v \sum_{v} \text{when } x = v \text{ do } (\text{tell}(\text{change}(x)</td>
</tr>
<tr>
<td>$\text{exch}_{q}[x, y]$</td>
<td>local $v \sum_{v} \text{when } t = v \text{ do } (\text{tell}(\text{change}(x)</td>
</tr>
</tbody>
</table>

Table 2.3: Cell Definition

**Recursive definitions**

Finally, a basic recursion can be defined in ntc with the form $q(x) \equiv P_{q}$, where $q$ is the process name and $P_{q}$ is restricted to call $q$ at most once and such call must be within the scope of a “next”. The reason of using “next” is that we do not want an infinite recursion within a time-unit. Recursion is used to model iteration and recursive definitions. For instance, using this basic recursion, it is possible to write a function to compute the factorial function.

**2.4 Generic Constraint development Environment (Gecode)**

Gecode is a constraint solving library written in C++. Gecode provides efficient state-of-art propagators for multiple constraints and configurable search-engines.

**2.4.1 Constraints as Propagator Agents (CPA)**

“Gecode is based on the constraints as propagator agents (CPA). CPA systems provide a propagator for each type of user defined constraint. Propagators translate a constraint into basic constraints supplying the same information. Basic (finite domain) constraints have the form $x \in [a..b]$. For instance, in a store (a set with all the constraints asserted) containing $pitch_1 \in [36..72], pitch_2 \in$
[60..80], a propagator for the constraint \( \text{pitch}_1 > \text{pitch}_2 + 2 \) would add constraints \( \text{pitch}_1 \in [63..72] \) and \( \text{pitch}_2 \in [60..69] \).

As described in the above example, the action of propagators ends up narrowing down the set of possible values for each variable. This, however, does not guarantee that it will eventually be inferred a single value to each variable. CPA systems thus include search engines. The purpose of a search engine is to choose additional basic constraints to add into the store until all variables have reduced their domain into a single value. Using them we can find one, many, or all the solutions for a CSP.\(^{35}\) The reader may notice that there is a similarity between the CPA and the ntcc models. Both of them are based on concurrent agents working over a constraint store.

2.4.2 Advantages

Gecode works on different operative systems and is currently being used as the constraint library for Alice\(^{33}\) and Mozart-Oz, therefore it is very likely to be maintained for a long time. Furthermore, it provides an extensible API, allowing us to create new propagators and user-defined search engines. For instance, we can extend Gecode to reason about trees and graphs. Finally, Gecode’s performance is better than the constraints solving tool-kits used in Sicstus Prolog and Mozart-Oz (according to Benchmarks presented in \url{http://www.gecode.org}).
Chapter 3

Developing real-time capable lightweight (lw) threads for Common Lisp

We explored a variety of strategies to implement lw threads in Common Lisp. Since OpenMusic [5] and Omax [4] are written in Common Lisp, developing lw threads will allow them to take advantage of lightweight concurrency on their applications.

OpenMusic and Omax are applications developed by Ircam in Common Lisp. Currently, they use Lispworks (a commercial Common Lisp distribution). In order to keep our interpreters compatible with OpenMusic and Omax, we have explored the possibility of implementing generic lightweight threads for Common Lisp, testing their performance in Lispworks Professional 5.02 under Mac OS X for Intel.

3.1 Using continuations

3.1.1 Motivation

It is possible to simulate concurrent threads using continuations and UNIX signals (to provide preemption). The continuation of each thread must be saved. This way they can be invoked at a later time. When a thread must block, we can capture the continuation using the call/cc macro (provided by libraries such as cl-cont) since continuations are not natively implemented in Common Lisp. This approach was used before to implement a concurrent version of ML called SML/NJ [8].

3.1.2 Disadvantages

Using continuations posses a few problems. They only capture the state that describes the processor, but they do not capture the state of the I/O systems [47]. Another issue is the lack of a native implementation in Common Lisp. Even though, the Continuation Passing Style (CPS) can be obtained writing Lisp macros, it creates an overhead leading to a high memory and time consumption.

3.1.3 Tests

Following, we present some tests comparing Lisp code with CPS code.

Adding the elements of a list

Figure 3.2 describes the time consumption of a function adding all the elements of a list containing 5000 and a list containing 20000 elements. We executed several times both the Lisp code and the CPS code (generated by cl-cont) in Lispworks and SBCL. Results were obtained after several
tests under Mac OS 10.5 using an Imac Intel Core 2 duo 2.8 ghz, Lispworks Professional 5.02, and SBCL 1.012. We concluded that Lispworks performance is not very good for CPS code, probably because the compiler is not optimized to handle CPS code. Therefore, we do not recommend using this approach to implement lightweight threads for Common Lisp. The reader should be aware that both the Lisp code and the CPS code were previously compiled.

Figure 3.1: Traversing a List of n elements. CSP done by using cl-cont.

3.2 Using Event-Driven Programming

3.2.1 Motivation

Each program is written as an event loop, which runs by taking an event and executing some code depending of the type of event, and then posting one or more new events. Lightweight threads can be implemented this way, having several event queues (one for each thread). The scheduler picks one event from one queue to execute each time around.

In order to achieve thread synchronization, we made a wait and a bind event working on top of dataflow variables. Asynchronous send and blocking receive can be achieved assigning a mailbox to each thread. The scheduler picks an event from one queue to execute each time around.

Our implementation of lightweight threads for Common Lisp is composed by: a runnable thread queue, current thread variable, and a hash table to keep a relation between a lock and the threads waiting for that lock. The threads are modelled by a structure containing an identifier, an status (suspended, running, terminated), a reference to a synchronization variable (when it is suspended), an event queue and a priority.

(defstruct thread name status whoamiwaitingfor EventQueue Priority)

We also provide 6 simple kind of events: execute, bind, wait, let, waitforlock and dotimes with a handler associated to each of them. The handler for the execute event is simple, it evals the instruction encapsulated in this event. Notice that this leaves the responsibility of using it only for simple instructions to the programmer. For instance, encapsulating an infinite loop or an infinite recursion inside this event leads to an unfair scheduling. The bind and the lock events are used for synchronization and the let and dotimes events help fragmenting blocks containing multiple instructions.

3.2.2 Disadvantages

Although transforming Common Lisp code to event driven programming came up being efficient, the lack of generality of this approach, makes it inappropriate for many applications. For instance, it will be necessary to create events for go to jumps, for exception handling, asynchronous signals, loop macros, complex programs.
3.2.3 Tests

We made some tests for the event-driven programming interface for Common Lisp. We compared it with Lisp processes and with a sequential program doing the same.

Concurrent matrix multiplication

The code below represents an implementation of a multithreaded matrix multiplication algorithm. For each multiplication necessary, a new thread is created \((n^3)\) threads are created for two square matrices of size \(n\) and synchronization is provided by locks. This is the implementation using Lispworks processes as the threading library

\[
\text{(defun runrunrun ()}
\let ((thelock (mp:make-lock)) (thelock2 (mp:make-lock)))
\let (dotimes (i n ) (dotimes (j n ) (dotimes (k n )
\let (mp:process-run-function "Cik = Aij*Bjk" nil)
\let (lambda (II JJ KK) (mp:process-lock thelock)
\let (setf (aref *C* II KK) (+ (aref *A* II JJ) (aref *B* JJ KK) )))
\let (mp:process-unlock thelock) (mp:process-lock thelock2)
\let (setf *counter* (- *counter* 1)) (mp:process-unlock thelock2)) i j k)))))
\]

Contrasting to the implementation above, our implementation uses the event driven programming interface described previously. Threads do not use locks, since each \textit{setf} instruction is made atomically with the \textit{execute} event. Instead, they use dataflow variables (having two states, bind or not bind) to be synchronized with another thread in charge of telling the user when the execution of all the threads is done.

Figure 3.2 compares the execution times of the Lispwork processes (native medium weight threads provided by Lispworks), our implementation of event driven programming and the sequential version of the matrix multiplication algorithm in an Intel 2.8 GHz using Mac OS 10.5.2, running Lispworks 5.02 professional. Additionally, we tested simple-processes provided by Lispworks multiprocessing API, but they were very slow, taking around 10 seconds for 16 threads. Furthermore, they are very unstable in Lispworks 5.0 under Mac OS X, often crashing the whole IDE.

Figure 3.2: Multithreaded matrix multiplication (time in seconds)
Chapter 4

Developing real-time capable interpreters for CCP

CCP is the predecessor of ntcc. CCP has been used to model a variety of systems. It is also the base of programming languages such as Mozart-Oz. For that reason, we started by developing interpreters for ccp and comparing their performance, before developing a real-time interpreter for ntcc.

In order to develop an interpreter for CCP, we need to provide a way to encode the ask processes, the tell processes and the parallel processes in Gecode. Ask processes can be easily represented in Gecode taking advantage of the reified propagators and tell processes can be represented with non-reified propagators. Additionally, it is important to mention that Gecode is not thread safe, being necessary to add locks for all the concurrent reading and writing operations, adding an overhead when using threads. Another fact is the event driven nature of Gecode itself [45], allowing us to express CPP and ntcc, without writing code for a dispatcher nor event queues. In this section, we will explain the different approaches explored to develop a generic real-time interpreter for CCP. Further details about how to encode processes as propagators are presented in next chapter.

4.1 Our previous approaches

We tried some combinations of programming languages (C++ and Common Lisp) and concurrency models (threads and event-driven programming).

4.1.1 Gecode interfaces to Common Lisp

The first problem we faced when designing the interpreter was interfacing Gecode to Common Lisp (since OpenMusic is written on Common Lisp). First, we redesigned the Gecol (an Opensource interface for Gecode 1.3.2 originally developed by Killian Sprote) library to work with Gecode 2.2.0 (current version of Gecode). Unfortunately, Gecol 2 is still a low-level API as Gecol. For that reason, using it requires deep knowledge of Gecode and it has a difficult syntax. To fix that inconvenient, we decided to upgrade the Gelisp library to Gecode 2.2.0 [35], originally developed by Rueda for Gecode 1.3.2. We successfully used this library to solve Constraint Satisfaction Problems (CSP) in the computer music domain. This library is easier to use and could be the foundation of a new version of the interpreter. Both, Gecol and Gecol 2 can be found at http://common-lisp.net/project/gecol/.

4.1.2 Threads in Lisp and C++

Using Gecol 2, we developed a prototype for the ntcc interpreter in Lispworks 5.0.1 professional using Lispworks processes (based on pthreads) under Mac OS X. In a similar way, we made another interpreter using C++, Gecode, and Pthreads (a portable implementation of medium weight
threads in C) for concurrency control. In both threaded prototypes the tell agents are modeled as threads waiting until the store is free, which then add a new constraint to the store.

On the other hand, the when processes are threads waiting until the store is free and asking if their condition can be deduced from the store. If they can deduce its condition they execute their continuation, else they keep asking (see figure 4.1). The conditions for the when processes are represented by boolean variables linked to reified propagators (recall that $C \leftrightarrow b$ is a reified propagator for the constraint C). Fortunately, Gecode provides reified propagators for most constraints used in multimedia interaction (e.g., equality and boolean constraints).

![Threaded ntcc interpreters using Lispworks and using C++](image)

Figure 4.1: Threaded ntcc interpreters using Lispworks and using C++

Since Gecode is not thread-safe (it does not support the concurrent access to its variables and functions), we protect the access to a Gecode Space with a lock, synchronizing the access to Gecode. However, we still have a problem. Each time we want to ask if a condition can be deduced from the Store, we call Gecode’s status function (a Gecode function used to calculate a fixpoint), because propagators in Gecode are “lazy” (they only act by demand). The drawback of both threaded implementations (in C++ and Lispworks) is the inefficiency of using the status function each time they want to query if the “when” condition can be deduced. Making extensive use of the status function would be inefficient even if we use an efficient lightweight threads library such as Boost (http://www.boost.org) for C++.

4.1.3 Event-driven programming in Lisp

After discarding the threading model, we found a concurrency model giving us better performance. We chose event-driven programming for the implementation of the next prototype. This model is good for a ntcc interpreter because we do not use synchronous I/O operations and all the synchronization is made by the ask processes (when, $\sum$, and Unless) using constraint entailment.

This prototype works on a very simple way. There is an event queue for the ntcc processes, the processes are represented by events, and there is a dispatcher handling the events. The handler for the When events checks if the boolean variable $b$, representing their waiting condition, is assigned. If it is not assigned, it adds the same When event to the queue, else it checks the value of $b$. If $b$ is true, it adds the continuation of the When events to the event queue, otherwise no actions are taken. On the other hand, the handler for Tell events add a constraint to the store. The store is represented as a Gecode Space. Finally, the handler for the Parallel events adds all its sub-processes to the event queue (see figure 4.2).

Using event-driven programming led us to a faster and easier implementation of ntcc than the approaches presented before. However, we realized that instead of creating handlers for tell, ask, and parallel; and a dispatcher for processing the events, we could improve the interpreter’s performance taking advantage of the dispatcher and event queues provided by Gecode for scheduling its propagators.
4.2 Our solution: Encoding processes as Gecode propagators

After considering multiple solutions, we found out that encoding processes as Gecode propagators is a generic implementation of the CCP interpreter capable of real-time. The tell, ask and parallel processes are represented by classes.

We defined an AskBody class, which is a superclass for the tell, ask and parallel classes. This way we can pass any object inhering from this class to the ask propagator, making it generic. We do not use function pointers, because then it would be also required to pass the arguments to those functions and it will be less generic.

We also defined an interface (the superclass tell) and three classes inhering from it: tellEqual, representing \(a = b\); tellSetIn, representing \(a \in B\); and tellGE, representing \(a > b\). Other kind of tell agents can be easily extended inheriting from the tell superclass and declaring an Execute method. The Execute method is called by an ask object when a tell is nested in an ask or it is called by a parallel object when it is nested in a parallel object.

In order to represent the ask processes, we have developed a generic ask class, with a constructor receiving a pointer to an AskBody object and a pointer to a constraint object. Both of them are passed to the ask propagator, when its Execute method is called. The AskBody object \(P\) is the continuation of the ask and the constraint object \(b\) is the ask guard.

These classes inherit from the constraint class: SetIn for \(a \in B \iff b\), EQ for \(a = c \iff b\), GQ for \(a \geq c \iff b\), GE for \(a > c \iff b\), NOT for \(\neg(a) \iff b\), AND for \(a \land c \iff b\) and OR for \(a \lor c \iff b\). This can also be extended by inheriting from the constraint class and declaring a `get_boolean()` method, which returns a GECODE Boolean variable.

Once \(b\) is assigned, the propagator checks its value. For a true value, it calls the Execute method of \(P\) (which could be another ask, a tell or a parallel). Then the ask propagator will go to the subsumed state.

```c
ExecStatus AskPropagator::propagate(Space* home, ModEventDelta med) {
    if (b.one()) {P->Execute(home); assert(b.assigned()); goto subsumed;}
    if (b.zero()) {assert(b.assigned()); goto subsumed;}
    return ES_FIX;
    subsumed:
    return ES_SUBSUMED(this,sizeof(*this));}
```

We compared different interpreters running the program to find, concurrently, a path in graph (fig. 4.3). We present the execution times of a Common LISP recursive function, an implementation using Concurrent Constraint Programming in Mozart-OZ, an implementation using our own dispatcher in Common LISP and the implementation in C++ using the ask propagator. The reader may notice that the performance of the interpreter using the ask propagator is much faster than all the other ones. Therefore, we recommend encoding processes as Gecode propagators for real-time applications using the CCP model.
4.3 Applications

4.3.1 Finding paths in a graph concurrently

An application where we use the CCP interpreter to define, concurrently, paths in a graph. The idea is having one CCP process for each edge. Each Edge\_Process\((i, j)\) sends forward “signals” to its successors and back “signals” to its predecessors. When an Edge\_Process\((i, j)\) receives a back “signal” and a forward “signal”, it tells the store that there is a path and adds \(j\) to the set \(next_i\) (A finite set variable containing the successors of the vertex \(i\)). After propagation finishes, we iterate over the resulting sets to find different paths. For instance, we can build a path in the graph getting the lower-bound of each set using the variable \(next_i\).

Formal definition

This process represents an edge in a graph.

\[
\text{Edge\_Process}(i, j) \overset{\text{def}}{=} \begin{cases} 
\text{when } \text{Forward}_i \land \text{Back}_j \text{ do } (\text{tell } (\text{it\_exists} = \text{true}) \parallel \text{tell } (j \in \text{Next}_i)) \\
\parallel \text{when } \text{Forward}_i \text{ do } \text{tell } (\text{Forward}_j = \text{true}) \\
\parallel \text{when } \text{Back}_j \text{ do } \text{tell } (\text{Back}_i = \text{true})
\end{cases}
\]

The Main process finds a path between the vertices \(a\) and \(b\) in a graph represented by edges (a set of pairs \((i, j)\) representing the graph edges). The Main process calls Edge\_Process\((i, j)\) for each \((i, j) \in \text{edges}\) and concurrently, it sends forward “signals” to processes with the form Edge\_Process\((a, j)\) and back “signals” to processes with the form Edge\_Process\((i, b)\). Notice that sending and receiving those “signals” is greatly simplified by using \text{tell}, \text{ask} and the CCP \text{store}.

\[
\text{Main}(\text{edges}, a, b) \overset{\text{def}}{=} \prod_{(i, j) \in \text{edges}} (\text{Edge\_Process}(i, j)) \\
\parallel \text{tell } \text{Forward}_a \text{ = } \text{true} \\
\parallel \text{tell } \text{Back}_b \text{ = } \text{true}
\]

Example

Following, we give an intuition about how this system works. To find a path between the vertices 1 and 5 in figure 4.4, it starts by sending forward “signals” to all the processes with the form Edge\_Process\((1, b)\) and back “signals” to all the processes with the form Edge\_Process\((a, 5)\). As soon as an Edge\_Process receives a back “signal” and a forward “signal”, it tells the store that there is path (i.e., \text{tell } (\text{it\_exists} = \text{true}) ).
Additionally, the reader may notice that there is not a path between vertices 1 and 5 in figure 4.5. In this example, the back “signals” sent to processes $Edge_{Process}(a,5)$ are not received by any process. Therefore, none of the $Edge_{Processes}$ receives a back and a forward signal.

After calculating a fix point, we can ask the constraint system for the value of \( it \text{exists} \). If the variable is not bounded, we can infer that there is not a path.
Chapter 5

Developing a real-time capable interpreter for ntcc

During the last decade, three interpreters for ntcc have been developed. Lman [24] by Hurtado and Muñoz in 2003, NtccSim (http://avispa.puj.edu.co) by the Avispa research group in 2006, and Rueda’s sim in 2006. They were designed to simulate ntcc models, but they were not made for real-time interaction.

The idea is not creating a new programming language based on ntcc. Our goal is creating interpreters on programming languages, taking advantage of the libraries (e.g., GUI or Midi/Audio processing) available for them.

When designing ntcc interpreters, we need to find a constraint solving library or programming language allowing us to check stability (i.e., know when a time-unit is over), check entailment (i.e., know if a constraint can be deduced from the store), post constraints, and synchronize the concurrent access to the store.

The authors of the ntcc model for interactive scores proposed to use Gecode as a constraint solving library for future ntcc interpreters and creating an interface for Gecode to OpenMusic. Furthermore, they propose extending Lman (only runs under Linux) to work under Mac OS X using Gecode.

5.1 History of ntcc Interpreters

5.1.1 Lman

Lman was developed as a framework to program RCX Lego Robots. It is composed of three parts: an Abstract machine [24], a compiler [27] and a visual language [14]. We took from this interpreter the idea of having several queues for storing Ntcc’s processes, instead of using threads. Regrettably, since Lman is implemented in the C language - which does not offer abstractions such as objects - its extension is difficult. Finally, it only supports finite domain constraints and it was not designed for real-time interaction.

5.1.2 Ntccsim

NtccSim was used to simulate biological models [17]. It was developed in Mozart-Oz [34]. It is able to work with finite domains (FD) and a constraint system to reason about real numbers. We conjecture (it has not been proved) that using Mozart-Oz for writing a Ntcc interpreter it is not as efficient as using Gecode, based on the results obtained in the benchmarks of Gecode, where Gecode performs better than Mozart-Oz in constraint solving.
5.1.3 Rueda's interpreter

Rueda's interpreter was developed as a framework to simulate multimedia semantic interaction applications. This interpreter was the first one representing rational trees, finite domain (FD), and finite domain sets (FS) constraint systems. One drawback of this interpreter is the use of Screamer [48] (framework for constraint logic programming written in Common Lisp) to represent the constraint systems. Unfortunately, Screamer is not designed for high performance. This makes the execution of the Ntcc specifications in Rueda's interpreter not suitable for real-time interaction.

5.2 Our Solution: Encoding ntcc processes as propagators

Our solution, is once again based on a simple but powerful concept. The when and ∑ processes are encoded as propagators in Gecode. That way Gecode manages all the concurrency required for the interpreter. Gecode calls their continuations when their conditions are assigned to true. On the other hand, tell processes are trivially codified to existing Gecode propagators and timed processes (i.e., those using the agents +, !, unless, ←, or next) are managed providing different process queues for each time-unit in the simulation. This prototype is called Ntccrt.

In this section we focus on describing the data structures required to represent each Ntcc agent. We also give a brief description of how some processes execute and how some constraint systems are modeled. Finally, we explain how the interpreter makes a simulation of a Ntcc specification. Ntcc agents are represented by classes. To avoid confusions, we write the agents with **bold font** (e.g., when C do P) and the classes with *italic font* (e.g., When class).

5.2.1 Finite Domain (FD), Finite Set (FS), and Infinite rational trees

To represent the constraint systems we need to provide new data types. Gecode variables work on a particular *Store*. Therefore, we need an abstraction to represent Ntcc variables present on multiple stores (one for each time-unit) with the same name. Boolean variables are represented by the *BoolV* class, FD variables by the *IntV* class, FS variables by the *SetV* class, and infinite rational trees with unary branching by *SetVArray*, *BoolVArray*, and *IntVArray* classes.

5.2.2 Representing the tell agent

After encoding the constraint systems, we defined a way to represent each process. All of them are classes inheriting from *AskBody*. *AskBody* is class, defining an *Execute* method, which can be called by another object when it is nested on it. To represent the tell agent, we defined a super class *Tell*. For this prototype, we provide three subclasses to represent these processes: *tell*(a = b), *tell*(a ∈ B), and *tell*(a > b). Other kind of tell agents can be easily defined by inheriting from the *Tell* superclass and declaring an *Execute* method.

5.2.3 Representing the when agent

For the when agent, we made a *When propagator* and a *When* class for calling the propagator. A process when C do P is represented by two propagators: C ↔ b (a reified propagator for the constraint C) and if b then P else skip (the When propagator). The When propagator checks the value of b. If the value of b is true, it calls the Execute method of P. Else, it does not take any action.

5.2.4 Representing the Non-deterministic agent

To represents the ∑ agent (i.e., non-deterministic choice) we made the parallel conditional propagator. This propagator receives a sequence of tuples [<b₁, P₁> ... <bₙ, Pₙ>], where bᵢ is a Gecode boolean variable representing the condition of a reified propagator (e.g., a = c ↔ bᵢ) and Pᵢ (a pointer to an AskBody object) is the process to be executed when bᵢ is assigned to true. The propagator executes the process Pᵢ associated to the first guard that is assigned to true. It means Pᵢ such that k = min({1 < i < n, bᵢ = true}). Then,
its work is over. If all the variables are assigned to false its work is over too. This propagator is based on the idea of the Parallel conditional combinator proposed by Schulte [43]. A curious reader might ask how we obtain a non-deterministic behavior. In order to make a non-deterministic choice, we pass the parameters to the propagator in a random order.

```c
ExecStatus ParallelConditional::propagate(Space* home, ModEventDelta med) {
    int falses = 0;
    for(int i=0; i < x.size(); i++)
        if (b[i].one()) {P[i]->Execute(home); goto subsumed;}
        else if (b[i].zero()) {assert(b[i].assigned()); falses++;}
    if (falses == b.size()) { goto subsumed; }
    return ES_NOFIX;
subsumed:
    return ES_SUBSUMED(this,sizeof(*this));}
```

5.2.5 Representing local variables

Local variables are easily represented by an instruction allowing the user to create a new “fresh” variable at the beginning of a procedure. Then, that new variable is going to persist during the following time-units being simulated. The other variables are declared at the beginning of the simulation.

5.2.6 Representing timed processes

Timed processes are represented by the TimedProcess class. It is an abstract class providing a pointer for the current time-unit, for a queue used for the unless processes, for a queue used for the persistent assignation processes, for a queue used for the other processes, and for the continuation process. Each subclass defines a different Execute method. For instance, the Execute method for the Star class randomly chooses the time-unit to place the continuation (an AskBody object) on its the corresponding process queue.

5.2.7 Representing the unless and persistent assignation agents

The Unless class and the Persistent assignation class are different. Their Execute methods are called only after calculating a fixpoint. Then, if the condition for the Unless cannot be deduced from the stable Store, its continuation is executed in the next time-unit. On the other hand, the Persistent assignation copies the domain $D$ of the variable assigned, when the Store is stable. Then it assigns $D$ to that variable in following time-units.

5.2.8 Representing ntcc definitions

We also have a Procedure class used to model both, Ntcc simple definitions (e.g., $A \overset{def}{=} \text{tell}(a = 2)$) and NTCC recursive definitions (e.g., $B(i) \overset{def}{=} B(i + 1)$), which are invoked using the Call class. For Ntcc recursive definitions, we create local variables simulating call-by-value (as it is specified in the formalism).

5.2.9 Execution model

In order to execute a simulation, the users write a Ntcc specification in C++, compile it, and then they call the compiled program with the number of units to be simulated and the parameters (if any) to the main Ntcc definition. For each time-unit $i$, the interpreter executes the following steps:

First, it creates a new NTCCSpace (which inherits from the Gecode space class). Then, it creates a new store and new variables in the store. Then, it process the input (e.g., Midi data coming from PD or Max). If $i = 0$, it calls the main Ntcc definition with the arguments given by the user.

After that, it moves the unless processes to the $i^{th}$ unless queue, moves the persistent assignation processes to the $i^{th}$ persistent assignation queue, and executes all the remaining processes in the $i^{th}$ process queue. Then, it calculates a fixpoint (using the status function). Note how we only call the status function each time-unit, opposed to the previous prototypes.
After calculating a fixpoint, it executes the unless processes in the $i$th unless queue and executes the persistent assignment in the $i$th persistent assignment queue. Then, it calls the output processing method (e.g., sending some variable values to the standard output or through a midi port). Finally, it deletes the current NTCCSpace.

Although we developed a portable, generic, and real-time capable interpreter for Ntcc; we still had a problem. In order to write a Ntcc specification, it was necessary to write code in C++ and then compiling it. This was clearly counter-intuitive for non-computer scientists. For that reason, we developed a parser on top of OpenMusic, where both computer scientists and musicians, can write Ntcc specifications in a graphical way. Every specification is automatically compiled as an stand-alone application using Midishare or as a external for Pd or Max.

In order to make an interface for OpenMusic, we developed a Lisp parser using Common Lisp macros to easily write an ntcc specification in Lisp syntax and translate it to C++ code. Those macros also automatically compile a Ntcc program. Then, we created OpenMusic methods (a graphical representation for Common Lisp methods using the CLOS system) to represent all ntcc processes, providing a mechanism to generate an input for the parser when the OpenMusic methods are evaluated.

Finally, to handle Midi streams (e.g., files, Midi instruments, or Midi streams from other programs) we use the predefined functions in Pd or Max to process Midi. Then, we connect the output of those functions to the Ntccrt binary plugin. We also provide an interface for Midishare, useful when running stand-alone programs.

5.3 Applications

5.3.1 The dining philosophers

Synchronization of multiple operations is not an easy task. For instance, consider the problem of the dining philosophers proposed by Edsger Dijkstra. It consists of $n$ philosophers sitting on a circular table and $n$ chopsticks located between each of them. Each philosopher, is thinking until it gets hungry. Once he gets hungry, he has to take control of the chopsticks to his immediate left and right to eat. When he is done eating, he restarts thinking.

The dining philosophers problem mentioned before, has a few constraints. The philosophers cannot talk between them and they require both chopsticks to eat. Furthermore, a solution to this problem must not allow deadlocks, which could happen when all the philosophers take a chopstick and wait forever until the other chopstick is released. Additionally, it must not allow starvation, which could happen if one or more philosophers are never able to eat.

We propose a solution to this problem for $n$ philosophers, using the Ntcc formalism. All the synchronization is made by reasoning about information that can be entailed (i.e., deduced) from the store or information that cannot be deduced (using the unless agent). This way, we can have a very simple model of this problem on which the synchronization is made declarative.

The recursive definition $\text{Philosopher}(i, n)$ represents a philosopher living forever. The philosopher can be in three different states: thinking, hungry or eating. When the philosopher is on the thinking or eating state, it will choose non-deterministically to change to the next state or remain on the same state in the next time-unit. On the other hand, when the philosopher is on the hungry state, it will wait until he can control the first (F) chopstick (left for even numbered and right for odd numbered). As soon as he controls the first chopstick, it will wait until he can control the second (S) chopstick. Once he controls both chopsticks, it will change to the eating state in next time unit.

Formal definition

\[
\text{Philosopher}(i, n) \overset{\text{def}}{=} \\
\text{when } St_i = \text{thinking do next} \\
\quad (\text{tell } (St_i = \text{hungry}) + \text{tell } (St_i = \text{thinking})) \\
\| \text{when } St_i = \text{hungry do} \\
\quad \text{when } ctrl_F = i \text{ do} \\
\quad \quad \text{when } ctrl_S = i \text{ do next}
\]
(tell \( St_i = eating \)) \( \parallel \) (tell \( ctrl_S = i \)) \( \parallel \) (tell \( ctrl_F = i \))
\( \parallel \) unless \( ctrl_S = i \) next
(tell \( i \in wait_S \)) \( \parallel \) (tell \( ctrl_F = i \)) \( \parallel \) (tell \( St_i = thinking \))
\( \parallel \) unless \( ctrl_F = i \) next (tell \( i \in wait_F \)) \( \parallel \) (tell \( St_i = thinking \))
\( \parallel \) when \( St_i = eating \) do next
(tell \( St_i = thinking \)) \( \parallel \) (tell \( ctrl_S = i \)) \( \parallel \) (tell \( ctrl_F = i \)) \( \parallel \) (tell \( St_i = eating \))
\( \parallel \) when \( i \%2 = 0 \) do tell \( F = (i - 1)\%n \) \( \parallel \) (tell \( S = (i + 1)\%n \))
\( \parallel \) when \( i \%2 = 1 \) do tell \( F = (i + 1)\%n \) \( \parallel \) (tell \( S = (i - 1)\%n \))
\( \parallel \) next \( \) Philosophy(i,n)

The Chopstick\( (j) \) process chooses non- deterministically one of the philosophers waiting to control it, when the it is not being controlled by a process.

\( Chopstick(j) \) \( \stackrel{def}{=} \)
unless \( ctrl_j > -1 \) next
\( \sum_{x \in \text{Philosophers}} \) when \( x \in \text{wait}_j \) donext (tell \( ctrl_j = x \))
\( \parallel \) next \( Chopstick(j) \))

Finally, the system is modelled as \( n \) philosophers and \( n \) chopsticks running in parallel. The philosophers start their lives in the thinking state and all the chopsticks are free.

\( System(n) \) \( \stackrel{def}{=} \)
\( \prod_{i=0}^{n} \) (Philosopher\( (i) \) \( \parallel \) Chopstick\( (i) \) \( \parallel \) \( St_i = thinking \) \( \parallel \) \( ctrl_i = -1 \))

Implementation

Figure 5.1 shows a \( Pd \) program where the philosophers are represented as bangs (a graphical object design to send a message when the user clicks over it or when it receives a message from another object) and the concurrency control is made by a \textit{Ntccrt external}. When the philosophers start eating, the \textit{Ntccrt external} sends a message to the \textit{bang} changing its color. Chopsticks are represented as commentaries for simplicity.

![Figure 5.1: Synchronizing the dining philosophers using a \textit{Ntccrt external} in \textit{Pd}](image)

5.3.2 CCFOMI: Music Improvisation

Machine improvisation and related style simulation problems usually consider building representations of time-based media data, such as music, either by explicit coding of rules or applying
machine learning methods. For machine improvisation it is necessary to perform two activities concurrently: *Stylistic learning* and *Stylistic simulation*.

We call *Stylistic learning* the process of applying such methods to musical sequences in order to capture salient musical features and organize these features into a model. The *Stylistic simulation* process produces musical sequences stylistically consistent with the learned material [36].

A Concurrent Constraint Factor Oracle Model for Music Improvisation (CCFOMI) uses the *Factor Oracle* (FO) to store the information of the learned sequences and the *Ntec* formalism to synchronize both phases of the improvisation concurrently. FO is a finite state automaton constructed in linear time and space. It has two kind of transitions. *Factor links* are going forward and following them is possible to recognize at least all the factors from a sequence. *Suffix links* are going backwards and they connect repeated patterns of the sequence. Further formal definitions can be found in [1].

**Formal definition**

CCFOMI is divided in three subsystems: learning (ADD), improvisation (IMPROV) and playing (PLAYER) running concurrently. In addition, there is a synchronization process (SYNC) in charge of synchronization.

It has three kind of variables to represent the partially built Factor Oracle automaton: Variables \( s \) are the set of labels of all currently existing factor links going forward from \( k \). Variables \( S_i \) are suffix (i.e., backward) links from each state \( i \) and variable \( \delta_{k,i} \) give the state reached from \( k \) by following a factor link labeled \( \sigma \).

The variables \( s \) and \( \delta_{k,i} \) are modelled as rational trees, allowing us to add elements to them each time unit. For instance, with the constraints \( cons(A,B) \), \( cons(B,C) \), and \( cons(C,D) \) we can have a list of three elements \( [A,B,C,\ldots] \) and then we can add more elements, adding constraints to the variable \( D \).

The ADD process is in charge of building the FO (this process models the learning phase) by creating the factor links, the suffix links and the automata transitions. The specification of this process can be found in [30]. The learning and the simulation phase must work concurrently. In order to achieve that, it is required that the simulation phase only takes place once the subgraph is completely built. The \( SYNC_i \) process is in charge of doing the synchronization between the simulation and the learning phase to preserve that property.

Synchronizing both phases is greatly simplified by the used of constraints. When a variable has no value, when processes depending on it are blocked. Therefore, the \( SYNC_i \) process is “waiting” until \( go \) is greater or equal than one which means that the process has played the note \( i \) and the ADD process can add a new symbol to the FO. The other condition \( S_{i-1} \geq 0 \) is because the first suffix link of the FO is equal -1 and it cannot be followed in the simulation phase.

\[
SYNC_i \overset{\text{def}}{=} \begin{cases} \text{when } S_{i-1} \geq -1 \land go \geq i & \text{do} \\ \text{(ADD}_i \| \text{next } SYNC_{i+1}) \end{cases}
\]

The \( i \) process has played the note \( i \) and the ADD process can add a new symbol to the FO. The other condition \( S_{i-1} \geq 0 \) is because the first suffix link of the FO is equal -1 and it cannot be followed in the simulation phase.

\[
IMPROV(k) \overset{\text{def}}{=} \begin{cases} \text{when } S_k = -1 & \text{do next } (\text{tell } (out = \sigma_{k+1})) \\ \| \text{if } S_k \geq 0 & \text{do next } ((\text{tell } (out = \sigma_{k+1}) \| \text{CHOICE}(k + 1)) + \sum_{\sigma \in \Sigma} \text{when } \sigma \in from_{s_k} & \text{do } (\text{tell } (out = \sigma) \| \text{CHOICE}(\delta_{s_k}, \sigma))) \end{cases}
\]

30
|| unless \( S_k \geq -10 \) next \( CHOICE(k) \)

A wait\(_n\) process is necessary to wait until \( n \) symbols have been learned and launch the \( IMPROV(k) \) process.

\[
Wait_n \overset{def}{=} \text{when } go = n \text { do } IMPROV(n) \parallel \text{unless } go = n \text { do next } Wait_n
\]

The system is modelled as the Player and the Sync process running in parallel with a process waiting until \( n \) symbols have been played to start the Improv process.

\[
System_n \overset{def}{=} \text{tell}(S_0 = -1) \parallel PLAYER_i \parallel SYNC_i \parallel Wait_n
\]

Implementation

Stand-alone application. NTCC procedures are written in the interpreter in a declarative and intuitive way. For each procedure in the model (e.g., \( SYNC_i \)) it is necessary to declare and instantiate a class inheriting from \( procedure \), where the \( Execute \) method is overloaded to receive the arguments and return the resulting process. To use the rational trees constraint system we use the \( create\_IntV \) method provided by the \( store \) class, allowing us to reference an element in the rational tree. For instance, the element in the position \( i - 1 \) of the variable \( S \) can be referenced as \( thestore->create\_IntV(S,i-1,h) \). Once the element is referenced, we use it as we would do with a FD variable (\( IntV \)). Following this intuitive syntax, the \( Sync_i \) process, in charge of the synchronization between the \( PLAYER_i \) and the \( ADD_i \) processes, is written as

\[
Gecode::Int::AskBody * syncp::Execute()(Space * h,vector<int> intparameters, vector<variable *> variableparameters ){
    
    int i = intparameters[0];
    return ntcc::parallelp( ntcc::whenp(ntcc::ANDc(ntcc::GQc(thestore->create\_IntV(S,i-1,h), -1), ntcc::GQc(go,i)), ntcc::parallelp(ntcc::callp(Add,i), ntcc::nextnp(ntcc::callp(Sync,i+1)))),
    ntcc::unlessp(ntcc::ANDc(ntcc::GQc(thestore->create\_IntV(S,i-1,h), -1), ntcc::GQc(go,i)), ntcc::nextnp(ntcc::callp(Sync,i))) );}
\]

An external for Pd. Rueda et al ran \( CCFOMI \) on their interpreter. They wrote Lisp macros to extend Lisp syntax for the definition of \( Ntcc \) processes. We provide a similar interface to write \( Ntcc \) processes in Lisp. Furthermore, \( CCFOMI \) definitions are written in \( Ntccrt \) in an intuitive way using OpenMusic.

For instance, the \( Sync_i \) process, in charge of the synchronization between the \( PLAYER_i \) and the \( ADD_i \) processes, is represented with a few boxes: one for \( parallel \) processes, one for the \( \leq \) condition, one for the \( = \) condition, and one for \( when \) and \( unless \) processes (see figure 5.2).

![Figure 5.2: Writing the \( Sync_i \) process in OpenMusic](image-url)
We successfully specified CCFOMI in OpenMusic and we ran it as an stand-alone program using Midishare. We also ran it as a PD plugin generated by Ntccrt. The plugin is connected to the midi-input, midi-output, and a clock (used for changing from a time-unit to the other). For simplicity, we generate a clock pulse for each note played by the user (fig. 5.3). In the same way, we could connect a Metronome object. Metronome is an object that creates a clock pulse with a fixed interval of time.

Figure 5.3: Running CCFOMI in Pure Data (PD)

5.3.3 Signal processing

Ntcc was used in the past as an audio processing framework [38]. In that work, Valencia and Rueda showed how this modelling formalism gives a compact and precise definition of audio stream systems. They argued that it is possible to model an audio system and prove temporal properties using the temporal logic associated to ntcc. They proposed that a ntcc each time-unit can be associated to processing the current sample of a sequential stream. Unfortunately in practice this is not possible since it will require to execute 44000 time units per second to process a 44 Khz audio stream. Additionally, it posses problems to find a constraint system appropriate for processing signals.

Another approach to give formal semantics to audio processing is the visual audio processing language Faust [26]. Faust semantics are based on an algebra of block diagrams. This gives a formal and precise meaning to the operation programed there. Faust has also been been interfaced with Pd [15].

Our approach is different, we use a Ntcc program as an external for Pd or Max to synchronize the graphical objects in charge of audio, video or midi processing in Pd. For instance, the Ntcc external decides when triggering a graphical object in charge of applying a delay filter to an audio stream and it will not allow other graphical objects to apply a filter on that audio stream, until the delay filter finishes its work.

To illustrate this idea, consider a system composed by a collection of n processes (graphical objects applying filters) and m objects (midi, audio or video streams). When a process $P_i$ is working on an object $m_j$, another process cannot work on $m_j$ until $P_i$ is done. A process $P_i$ is activated when a condition over its input is true.

The system variables are: work$_j$ represents the identifier of the process working on the object $j$. end$_j$ represents when the object $j$ has finished its work. Values for end$_j$ are updated each time unit with information from the environment. input$_j$ represents the conditions necessary to launch process $i$, based on information received from the environment. Finally, wait$_j$ represents the set of processes waiting to work on the object $j$.

Objects are represented by the IdleObject($(j)$) and BusyObject($(j)$) definitions. An object is idle until it non- deterministically chooses a process from the wait$_j$ variable. After that, it will remain busy until the end$_j$ constraint can be deduced from the store.
Formal definition

\[ IdleObject(j) \stackrel{def}{=} \begin{cases} \text{when } work_j > 0 \text{ do next } BusyObject(j) \\ \text{∥ unless } work_j > 0 \text{ next } IdleObject(j) \\ \sum_{x \in P} \text{ when } x \in wait_j \text{ do tell } work_j = x \end{cases} \]

\[ BusyObject(j) \stackrel{def}{=} \begin{cases} \text{when } end_j \text{ do IdleObject}(j) \text{ ∥ unless } end_j \text{ next } BusyObject(j) \end{cases} \]

A process \( i \) working on object \( j \) is represented by the following definitions. A process is idle until it can deduce (based on information from the environment) that \( input_i \).

\[ IdleProcess(i, j) \stackrel{def}{=} \begin{cases} \text{when } input_i \text{ do WaitProcess}(i, j) \text{ ∥ unless } input_i \text{ next } IdleProcess(i, j) \end{cases} \]

A process is waiting when the information for launching it can be deduced from the store. When it can control the object, it goes to the busy state.

\[ WaitingProcess(i, j) \stackrel{def}{=} \begin{cases} \text{when } work_j = i \text{ do BusyProcess}(i, j) \text{ ∥ unless } work_j = i \text{ next } \text{ WaitingProcess}(i, j) \text{ ∥ tell } i \in wait_j \end{cases} \]

A process is busy until it can deduce (based on information from the environment) that the process finished working on the object associated to it.

\[ BusyProcess(i, j) \stackrel{def}{=} \begin{cases} \text{when } end_j \text{ do IdleProcess}(i, j) \text{ ∥ unless } end_j \text{ next } BusyProcess(i, j) \end{cases} \]

This systems models a situation with 2 objects and 4 processes. The implementation of this external can be adapted to any kind of objects and processes, represented by graphical objects in Pd. Ntcc only triggers the execution of each process \( work_j = i \), receives an input \( end_j \) when the process is done and another input \( input_i \) when the conditions to execute the process \( i \) are satisfied.

\[ System() \stackrel{def}{=} IdleObject(1) \text{ ∥ IdleObject}(2) \text{ ∥ IdleProcess}(1, 1) \text{ ∥ IdleProcess}(1, 2) \text{ ∥ IdleProcess}(2, 1) \text{ ∥ IdleProcess}(2, 2) \]

Implementation

This system is described in OpenMusic using the graphical boxes we provide. For this system, we use the ntcinbansignal to represent the bang inputs to the external in pd or max. (see fig. 5.4).
Figure 5.4: Writing a synchronization \textit{Ntcrt external} in \textit{OpenMusic}
Chapter 6

Developing libraries to solve musical CSP’s in Common Lisp

Gecode is a very efficient constraint solving library for C++. We are interested in developing an interface for that library to Common Lisp. First, we extended the Gecol library to work with current version of Gecode. Then, we realized that we needed a high-level API, because Gecol only provide a low level API to call Gecode functions directly. For that reason, we decided to extend the Gelisp library to work with current version of Gecode. Furthermore, we provided a graphical interface to represent CSP’s’s using OpenMusic and Gelisp.

6.1 Our previous approach: Extending Gecol

GECOL is a wrapper for Gecode 1.3 versions maintained by Killian Sprotte, providing propagators for finite domain (FD), finite domain sets (FS), the Deep-First-Search (DFS) and Branch-and-Bound (BAB) search engines. Gecol 2, the library we have developed, is an extension of Gecol maintained by Mauricio Toro Bermúdez, supporting Gecode 2.1.1 (current version of Gecode) and including further support for FS constraints. Gecode 2 is a low level API wrapping the propagators and the search engines mentioned before.

In order to write a finite domain CSP in Gecol 2, it is required to create a gecolspace (a class inheriting from Gecode’s space), declaring the number of variables to be used and their the domain. Then we add the constraints and specify the branching.

Gecode 2 vs Gecol 2

We wrote two benchmark examples provided by Gecode 2 in Gecol 2, the n-queens and all-distinct stress examples. The efficient version of n-queens, using all distinct constraints, was tested in both libraries in an Intel 2.8 GHz using Mac OS 10.5.2, Gcc 4.1, Gecode 2.1.1 and Lispworks 5.02 professional. The reader can notice (fig. 6.1) that time consumption of Gecol 2 is only about 50% more when using Gecode 2. On the other hand the memory consumption, presented in figure 6.2, is the around twice compared with Gecode 2.

Figure 6.1: Comparing Nqueens in GECODE and GECOL 2 (time in seconds)
Tests
We represented a Klumpenhouwer network (k-net) in Gecode and Gecol 2 as an adjacency matrix (a common representation for graphs). Following this representation, we wrote a CSP to find all the k-nets for a pitch class. First, we wrote a program in C++ using Gecode and then, in Common Lisp using Gecol 2 and Lispworks CAPI library (for drawing graphs). Gecode 2 runs around 3 times faster Gecol 2 for solving this problem, when we print the solutions. On the other hand, if we use Lisp lists to store all the solutions, time consumption and memory consumption gets very high using Gecol 2.

Klumpenhouwer networks (k-nets)

Transformational theory is an extension of classic American music set theory, which offers a formalized, mathematical approach to music analysis. The transformational approach, as it is explained by Hascher ([18]), arises from a simple questioning: Let \( a \) and \( b \) be two musical objects, what do we need to do to \( a \) in order to obtain \( b \)? The notions of transformational theory belong principally to group theory, as opposed to “mathematical” set theory on which “musical” set theory is based.

A Klumpenhouwer network (k-net) is a connected, valued, and directed graph, whose vertices are pitch classes, and whose edges are the operations of transposition \( t_m \) and inversion \( i_n \). To explain the intuition of transposition and inversions: let \( a, b \) be two pitches or elements of the set \( \{ C, C\#, D, ... B \} \). A transposition \( a \ t_m \ b \) means that \( (a + m) \mod 12 = b \). On the other hand, an inversion \( a \ i_n \ b \) means that \( b \) can be obtained from \( a \) “reflecting” \( a \) according to the \( n \text{th} \) symmetry line in a pitch circle (see figure ??). For instance, we can find different \( k \)-nets for the Pitch class \( \{ B, F\#, A \} \) as we can see in figure 6.3.

Formal definition

Formally, a CSP is a tuple \(< X, D, C >\) where \( X \) is the set of variables, \( D \) is a domain of values and \( C \) is the set of constraints. The input of this problem is a class pitch \( I \) represented as a tuple \(< i_1, i_2, ... i_n >\) and \( K \) the desired inversions. The variables for the CSP are \( X = < x_1, x_2, ... x_{n^2} >\), their domains are \( D = \{ 0, 1, 2 \} \). For the domain we represent when there is not an edge as 0, transpositions as 1 and inversions as 2.
For the constraints, we consider that if there is a transposition or inversion from \( i \) to \( j \) there is also one from \( j \) to \( i \), that way we can represent multiple solutions in a single adjacency matrix.

The constraints \( C \) are the following relations over all the variables in \( X \):

- the number of variables distinct from 0 are greater or equal than \( 2 \times n \)
- the number of variables equal to 2 are \( 2 \times K \)
- for each \( i \in [0..n], j \in [0..n] \) if \( i = j \) then \( x_{i+n+j} = 0 \)
- for each \( i \in [1..n], j \in [1..n] \) if \( i <> j \) then \( x_{i+n+j} = x_{j+n+i} \).

The “for each” constraints can be easily represented in Gecol 2 as follows:

```lisp
(dotimes (i n) (dotimes (j n)
  (if (equal i j)
    (gecol:rel... s (gecol:gecolspace-getint-int s (+ (* i n) j)) :irt-= 0 :icl-def)
    (gecol:rel... s (gecol:gecolspace-getint-int s (+ (* i n) j))
      :irt-= (gecol:gecolspace-getint-int s (+ i (* j n))) :icl-def))))
```

### 6.2 Our solution: Extending Gelisp

Gelisp provides an interface for Common Lisp and a graphical interface for OM. The syntax and the way how constraints are posted is greatly simplified (compared to Gecode) by using lists. The arrays of variables in Gecode are represented as lists in Lisp, allowing the user to apply list functions to them.

#### 6.2.1 Interface for Common Lisp

To solve a problem using this interface, we need to write a script. A script is a function to: construct a Computational Space (CS), define problem variables belonging to the CS and determine their domains, post constraints on the variables, and setup a search strategy. A CS comprises a store containing asserted constraints and a set of propagators interacting with it.

This interface allows the user to call most of Gecode propagators for both, Finite Domain (FD) and Finite Set (FS) constraint systems. We provide general constraints that are compiled to different Gecode methods according to the parameters given. For instance, \((\ < \ g \ (+g \ X \ Y \ Z) \ W)\) and \((\ > \ (\ g \ (+g \ X \ Y \ Z) \ 2)\) are compiled to different methods.

**Constraints for Finite Domain (FD)**

We provide FD propagators for: defining domains (e.g., \( \text{Domain}(X) = [2, 5] \)), arithmetic expressions (e.g., \( X + Y + Z \)), equalities and inequalities (e.g., \( X + Y < Z \)), sortedness and distinctness, minimum and maximum, cardinality (e.g., 1 occurs 2 times in \( [XYZ] \)), boolean constraints, and regular expression constraints.

**Constraints for Finite Set (FS)**

On the other hand, for FS we provide constraints for: defining domains (e.g., \( V \subseteq \{1, 2, 3\} \)), set expressions (e.g., \( A \cup B = C \)), set relations (e.g., \( X \subset V \)), set distinctness, and linking FD with FS variables.

**Performing search**

In addition, Gelisp includes two search engines, Deep Search First (DSF) and Branch-and-bound (BAB). The DFS engine works by choosing some variable, then a value for that variable, if this does not succeed (a constraint does not hold) then chooses another value. If the value succeed, then chooses another variable, then a value for it, etc.

The BAB engine works in a similar way but solutions are computed in such a way that each subsequent solution increases the value of some user specified FD variable. Both engines can be used for both FS and FD. In addition, we can parametrize heuristics for value and variable order.
Performing propagation

Furthermore, it is possible to execute propagation and observe the domain of the variables after
propagation. This is useful, for instance, to post temporal relations over musical objects and
observe the possible positions in time for each object[2].

Limitations

We do not provide constraints for the complete set representation, an efficient representation for
sets; the reflection API, used to get detailed information of the search and propagation; nor to
handle Gecode exceptions. We plan to have those features in next version.

6.2.2 Graphical Interface for OpenMusic

Instead of writing a script, in the graphical interface we represent a program with a special patch. A
patch is a visual algorithm, in which boxes represent functional calls, and connections are functional
compositions. Inside this CSP patch, we can place special boxes: to connect each constraint in the
CSP, to define variable and value heuristics, to define a time limit in the search, to connect the list
of variables that we want to observe, and a box to connect the variable to be optimized during the
search.

Representing constraints with graphical boxes

Furthermore, we provide a variety of boxes to represent simple constraints (e.g., $a = b$ and $a < 2$)
and high-level constraints (e.g., “all the intervals from a sequence must be different”). The output
of a CSP patch can be connected to three different kind of boxes: to find one solution, to find all
the solutions, and to perform propagation without search.

Novelties of the graphical interface

Using the graphical interface we can express problems declaratively with high-level constraints, but
unfortunately, some problems cannot be represented with the high-level constraints and require a
modeling using simple constraints and loops.

The high-level constraints can be parametrized. For instance, the graphical box to find the
intervals of a list ($x \rightarrow dx$) can be parametrized to find absolute, non-absolute, or modulo $n$
intervals. Additionally, it is possible to setup a parameter to post an all-distinct (i.e., the elements
of the list are pairwise different) constraint over the intervals.

6.3 Applications

Following, we describe both, an intuitive and formal definition of two CSP’s and we explain how
we solved them with Gelisp. Formally, a CSP is triple $<X, D, C>$, where $X$ is a set of variables,
$D$ are the domain values for each variable, and $C$ is a set of constraints (read as conjunction) over
the variables.

6.3.1 All-interval series

This problem can be generalized to find $n$ different notes with $n$ different inversional equivalent
intervals\(^1\) (including $V_n - V_0$). For instance, a value of $n = 24$ represents the all-interval series for
microtones.

\(^1\)For instance, an interval C-E is equivalent to E-C.
Formal definition

We formally define this CSP for an input $n$, as $n$ different variables with domain $[1..n]$, where all modulo $n$ intervals are pairwise different.

Variables: $V_1 \ldots V_n$
Domains: $[1..n] \ldots [1..n]$

Constraints:
- $C_1$ alldiff($V$)
- $C_2$ alldiff(($V_{i+1} - V_i$)$\%n$), $i \leq n$

We do not need to post a constraint for the interval $(V_n - V_0)$ because that interval is always 6, according to the literature. Furthermore, we know that it is enough to calculate the series where $V_0 = 0$, because the other ones can be obtained from that one. In addition, we know that if $V_1..V_n$ is an all-interval serie, $V_n...V_1$ is also. For that reason we model this two constraints to avoid symmetrical solutions:
- $C_3$ $V_0 = 0$
- $C_4$ $V_0 < V_n$

Graphical representation

We represent graphically this CSP with: a box to create the pairwise different variables, an $x \rightarrow dx$ for $C_2$ box, an equality for $C_3$, and inequality box for $C_4$.

Related work

Since the problem is about finding inversional equivalent intervals, previous attempts to solve this problem used an absolute value constraint to model $C_2$. That approach is not very efficient, because the absolute value cannot be expressed as a linear constraint, however the modulo $n$ constraint can.

6.3.2 Michael Jarrell’s CSP

Composer Michael Jarrel proposed a CSP for automatic music generation [19]. The goal is to generate $n$ notes. The notes have two type of segmentation, for the chords and for the motives. Each note belongs to a chord (depending on which segment the note is).

In addition, there are some motives and their desired amount for the intervals of each motives segment. Moreover, the first and the last note of the sequence are fixed. Finally, it is possible to have absolute or non-absolute intervals for the motives and allowing octaviation\(^2\) for the chords, the limits, or the motives.

Formal definition

Following, we define formally the CSP for the case of non-absolute intervals. For simplicity (in the formal description), we do not include octaviation nor segmentation.

Inputs:
- Motives $[M_1..M_A]$
- Occurrences $[OM_1..OM_A]$
- Chord $C$
- Limits $L_1$, $L_2$

Variables: $V_1 \ldots V_n$
Domains: $[0..127] \ldots [0..127]$

Constraints:

\(^2\)For instance, using octaviation, a pitch 62 (in Midi format) is equivalent to 50, 74, 86, etc.
\[ C_1 \forall 1 < i < n \ | \{ M_i, M_i \text{ is a subsequence of } \{ V_{i+1} - V_i, i < n \} \} | = OM_i \]
\[ C_2 \forall 1 < i < n \ Dom(V_i) = C \]
\[ C_3 \ V_1 = L_1 \land V_n = L_2 \]

Graphical representation

The graphical representation is composed by a few graphical boxes, without representing loops explicitly. In figure 6.4, we present the constraint \( C_1 \). Note how we use map iterators, the \( x \rightarrow dx \), and \( \text{motives-occurs=} \) boxes to find the intervals of each motives segment and to say how many occurrences of the motives are, respectively.

Figure 6.4: Constraint \( C_1 \) of Jarrell’s CSP

Related work

A previous attempt to solve this problem used OmBacktrack. Unfortunately, that library is no longer available in current version of OM.
Chapter 7

Concluding Remarks

We conclude this report by summarizing some concluding remarks from the previous chapters and presenting future research.

7.1 Results

“John McLaughlin, said to be one of the fastest Jazz guitarists, and found a minimum inter onset time of about 60 milliseconds. This figure gives an approximate constraint for the computation time of our system: it should be able to learn and produce sequences in less than 30 milliseconds.” According to the authors of the Continuator, a well-known machine improvisation software [28].

We ran CCFOMI in Ntccrt over an Intel 2.8 GHz using Mac OS 10.5.2 and GCC 4.1, taking an average of 20 milliseconds per time-unit, scheduling around 880 processes per time-unit, and simulating 300 time-units. Since we are learning and producing sequences with an answer time less than 30 milliseconds then, according to the authors of the Continuator, we have a system fast enough to interact with a musician.

7.2 Summary

- Using continuations in Lispworks is not very efficient because they do not work close to the compiler. An efficient implementation of lightweight threads in Common Lisp depends on the applications using the threads. For instance, for a CCP interpreter using Gecode, event-driven programming seems very natural, but for the Omax system, it could not be appropriate.

- On the other hand, a novelty of Ntccrt is the simplicity to represent concurrency and the iconic language designed to write the specifications, allowing non-computer scientists to easily model their systems. This interpreter can also represent processes that are not available in the formalism. Ntccrt offers two features not found in the Ntcc formalism. First, it is able to express general recursion (e.g., it can make multiple recursive calls in a recursive procedure), while the NTCC formalism offers a restricted kind of recursion. Second, since we encoded the When processes as a Gecode propagators, we are able to use search in Ntcc models without using the \sum agent. Models using non-deterministic choices are incompatible with the recomputation used in the search engines. This is not possible when encoding the when processes as threads.

- Unfortunately, the interpreter is not able to execute processes leading the Store to false. For instance,

\[
\text{when } false \text{ do next tell (fail = true) } \\
||\text{tell (a = 2)||tell (a = 3)}
\]
Since the **when** agent is represented as a propagator, once the propagation achieves a fail state no more propagators will be called in that time-unit, causing inconsistencies in the rest of the simulation. Fortunately, processes reasoning about a false Store can be rewritten in a different way, avoiding this kind of situations. For instance, the process above can be rewritten as

\[
\texttt{when state = false do next \; tell (fail = true)} \parallel \texttt{tell (state = false)}
\]

- Although Gecode was design for solving combinatory problems using constraints, we found out that using Gecode for Ntccrt gives us outstanding results for real-time. On the other hand, it is very expressive since most of the propagators used in real-time have a reified version, and those who does not have one, are easily extensible.

- Finally, we extended most Gelisp to work with OpenMusic. Our extensions provides an interface for most Gecode propagators and search engines. Furthermore, we provided some graphical boxes to represent constraints and search heuristics. Using Gelisp we can specify programs graphically and solve them almost as fast as using Gecode directly.

### 7.3 Future Directions

#### 7.3.1 Using a high-performance implementation of Common Lisp

In the future, in order to use *lightweight threads* in Common Lisp, we recommend exploring an implementation with *lightweight threads* such as CMU-CL ([http://www.cons.org/cmucl/](http://www.cons.org/cmucl/)). Note that current version of CMU-CL (CMU-CL 19e) provides binaries for Mac OS X.

#### 7.3.2 Applications for the CCP interpreter

We used the interpreter concurrently. It can be easily extended to find multiple paths in a bounded time, rank them according to a weight function, and returning the path with the highest rank. Since we represented the **ask** process as a monotonic propagator, we can use the Branch-and-Bound (BAB) search engine provided by Gecode, and the time objects (e.g., TimeStop) to manage the time demands. In the future, we propose using the CCP interpreter to find musical sequences in the Factor Oracle. This can be used in a music improvisation system such as Omax.

#### 7.3.3 Using Gelisp for Ntccrt

A problem arises when we want to call Lisp functions from the interpreter. Currently, we are only using Lisp to generate C++ code. However, it is not possible to embed Lisp code in the interpreter (e.g., calling a Lisp function as the continuation of a **when** process). To fix that inconvenient, we propose using **Gelisp** for writing a new interpreter, taking advantage of the call-back functions provided by the Foreign Function Interface (FFI) to call Lisp functions from C++. That way a process can trigger the execution of a lisp function.

#### 7.3.4 Adding support for cells for Ntccrt

The implementation of cells is still experimental and it is not yet usable. The idea for a real-time capable implementation of cells is extending the implementation of persistent assignation. Cells, in the same way than persistent assignation, require to pass the domain of a variable from the current time-unit to a future time-unit. However, persistent assignation usually involves simple equality relations. On the other hand, the cells assignation may involve any mathematical function \( g(x) \) (e.g., \( g(x) = x^2 - 2 \)).
7.3.5 Developing an interpreter for \texttt{pntcc}

There is an extension to make probabilistic choice in \texttt{ntcc}. The Probabilistic Non-deterministic Timed Concurrent Constraint (\texttt{pntcc} [29]) extends the non-deterministic choice with a probabilistic distribution. Probabilistic choice in \texttt{pntcc} is represented with the $\oplus$ operator.

Probabilistic choice is not yet possible in \texttt{Ntccrt}. For achieving it, we propose extending the idea used for non-deterministic choice agent $\sum$. To model $\sum$, it was enough to determine the first condition that can be deduced and then activate the process associated to it. For probabilistic choice, we need to check the conditions after calculating a fixpoint, because we need to know all the conditions that can be entailed before calculating the probabilistic distribution. When multiple probabilistic choice $\oplus$ operators are nested, we need to calculate a fixpoint for each nested level.

7.3.6 Developing an interpreter for \texttt{rtcc}

Finally, we found out that the time-units in \texttt{Ntccrt} do not represent discrete time-units, because in the simulation they have different durations. This is a problem when synchronizing an \texttt{ntcc} program with other programs. To fix it, we made the duration of each time-unit take a fixed time. We use a clock provided by \texttt{Pd} or \texttt{Max} and providing a clock input in \texttt{Ntccrt} plugins.

Unfortunately, there is not way a to describe the behavior of a \texttt{ntcc} time-unit if the fixed time is less than the time required to execute all the processes scheduled. For that reason we propose developing an interpreter for the \textit{Real Time Concurrent Constraint} (\texttt{rtcc}) [42] calculus. This calculus is an extension of \texttt{ntcc} capable of strong time-outs. Strong time-outs allows the execution of a process to be interrupted in the exact instant in which internal transitions cause a constraint to be inferred from the store. \texttt{Rtcc} is also capable of delays inside a single time unit. Delays inside a single time unit allows to express things like “this process must start 3 seconds after another starts”.

7.3.7 Adding other graphical interfaces for \texttt{Ntccrt}

For this work, we conducted all the tests under Mac OS X using \texttt{Pd}. Since we are using Gecode and \texttt{Flext} to generate the externals, they could be easily compiled to other platforms and for \texttt{Max}. We used Openmusic to define an iconic representation of \texttt{ntcc} specifications. In the future, we also propose exploring a way of making graphical specifications for \texttt{ntcc} similar to the graphical representation of data structures in \texttt{Pd}.

7.3.8 Developing model checking tools for \texttt{Ntccrt}

We propose using model checking tools for verifying properties complex systems. As far as we know, the only way to verify automatically \texttt{ntcc} and \texttt{pntcc} specifications is by running them on interpreters. For instance, we propose exploring the automatic generation of models for probabilistic model checker such as \textit{Prism}. The reader should be aware that \textit{Prism} has been used successfully to check properties of real-time systems [22].

7.3.9 Extending \texttt{Rules2Cp} for musical CSP’s in \texttt{Gelisp}

In addition, the idea of representing CSP’s and their heuristics with business rules from \texttt{Rules2Cp} [12] could be extended for music. The goal is writing rules, graphically, defining a musical CSP’s and simplifying the task of choosing heuristics manually.

7.3.10 Adding more features to search in \texttt{Gelisp}

We also want to represent \textit{recomputation} (a parameter for search engines in Gecode) graphically and include in \texttt{Gelisp} other methods to stop search (besides time limit), such as memory limit and failures limit, provided by Gecode.
Bibliography


45


