A Logic-based Language for Modeling and Verifying Musical Processes

Rafael Ramirez
Music Technology Group
Universitat Pompeu Fabra
Ocata 1, 08003 Barcelona, Spain
rafael@iua.upf.es

Abstract
In this paper we show how Tempo, a new logic-based language for high-level concurrent programming, can be used for modeling real musical processes. In particular, we show how Tempo allows the formal modeling of complex interactions among musical and audio processes. We argue that using Tempo for modeling musical processes provides an executable specification for musical problems and provides a formal programming framework in which the distribution of audio processes across several machines is transparent to the user. We also outline how model-based automatic verification methods can be directly applied to the formal verification of interesting properties of musical processes. We illustrate the language by presenting a number of musical examples.

1 Introduction
Music can be seen as constructed from complex interactions among a great number of concurrent processes acting on different musical dimensions. We share the view in (7) which considers musical structures such as rhythm and harmony as results of these interactions. Musical processes involve a wide variety of synchronization schemes requiring different degrees of precision about temporal and harmonic relations among them. The complexity of the different interactions of music processes poses great challenges to any computational formalism. In this paper we propose Tempo, a logic-based model for high-level concurrent programming, as a formalism to handle this complexity.

Tempo (2) is a high-level model of concurrency based on constraint entailment. The model provides a declarative formalism in which (a) the synchronization issues are treated as orthogonal to the system base functionality, and (b) distribution of audio processes in different machines is transparent to the user. In a musical context, this provides great advantages in specifying, manipulating and verifying musical processes (3).

The rest of the paper is organized as follows: Section 2 describes Tempo, the language used for modeling the interactions among musical processes and outline how Tempo specifications can be verified using model checking techniques. In Section 3 some musical examples are presented and finally Section 4 presents some conclusions and indicates some areas of future research.

2 Tempo
Time plays an extremely important role in music (5). Music can exist without pitch and other common musical features, but it is difficult to conceive music without time. Many researchers, e.g. (4; 8), have proposed methods for reasoning about temporal phenomena using partially ordered sets of events. Our approach to modeling musical processes is based on the same general idea. The basic idea here is to use a constraint logic program to represent the (usually infinite) set of constraints of interest. The constraints themselves are of the form $X < Y$, read as “$X$ precedes $Y$” or “the execution time of $X$ is less than the execution time of $Y$”, where $X$ and $Y$ are either events or positive integers, and $<$ is a partial order.

The constraint logic program (CLP) is defined as follows\(^1\).

Constants range over the set of positive integers and events classes $E, F, ...$ and there is a distinguished (postfixed) function $+$ which applies to events. Thus, the terms of interest, apart from variables and integers, are $e, e+, ..., f, f+, ...$

The idea is that $e$ represents the first event in the class $E$, $e+$ the next event, etc. Thus, for any event $X$, $X+$ is implicitly preceded by $X$, i.e. $X < X+$. We denote by $e(+N)$ the $N$-th event in the class $E$. Programs facts or predicate constraints are of the form $p(t_1, ..., t_n)$ where $p$ is a user de-

---

\(^1\)For a complete description, see (9; 10)
fined predicate symbol and \( t_i \) (\( 1 \leq i \leq n \)) are ground terms. Program rules or predicate definitions are of the form
\[
p(X_1, \ldots, X_n) \leftarrow B
\]
where the \( X_i \) are distinct variables and \( B \) is a rule body restricted to contain variables in \( \{ X_1, \ldots, X_n \} \). Variables prefixed by the symbol \# are variables of type integer. The rest of the variables are of type event. A program is a finite collection of rules and is used to define a family of partial orders over events. Intuitively, this family is obtained by unfolding the rules with facts indefinitely \(^2\), and collecting the (ground) precedence constraints of the form \( e < f \). For example, consider the following program with one rule for \( p \):
\[
p(e, f).
p(E, F) \leftarrow E < F, p(E+, F+).
\]
it defines the partial order \( e < f, e+ < f+, e++ < f++, \ldots \). Multiple rules for a given predicate symbol give rise to different partial orders. We will abbreviate the set of clauses:
\[
H \leftarrow C_{S_1}, \ldots, H \leftarrow C_{S_n}
\]
by the disjunction constraint \( H \leftarrow C_{S_1}; \ldots; C_{S_n} \) (disjunction is specified by the usual disjunction operator \(';\)\).

**Interpretation.** The constraint logic program as defined above has a procedural interpretation that allows a correct specification to be executed in the sense that processes run only as permitted by the constraints represented by the program. This procedural interpretation is based on an incremental execution of the program and a lazy generation of the corresponding partial orders. Constraints are generated by the CLP only when needed to reason about the execution times of current events (a detailed description of the interpreter and its Java implementation can be found in (10)).

**Process coordination.** Processes interact with each other by performing simple operations on a shared constraint store. Conceptually, processes can be distributed in several machines or executed concurrently in one machine. We have written both uniprocessor and distributed prototype Java implementations of the language which makes the issue of distribution transparent to the user. Processes can perform operations to check if the execution of a particular event is entailed by the constraints in the store, to write new objects into the store, or to read objects in the store. Synchronization is achieved only by using the check operation. The rest of the operations are non-blocking. (For a complete description of these operations see (11).)

\(^2\)We consider a musical system as an instance of a reactive concurrent system which, in general, is not required to terminate.

**Program verification.** Event annotations, i.e. check operations, naturally divide musical processes in a (often very small) number of states. Transitions among these states are explicitly determined by the system constraints. Taking into account these constraints it is quite simple to translate a musical system into a transition system. It is then straightforward to further translate the transition system of the musical system into a description in a model checker description language and directly apply the model checker to verify program properties. We have applied the SMV model checking system (6) to verify properties of our programs. We have implemented a system which automatically translates an extended Java program (i.e. a Java program synchronizing using Tempo constraints) into a model \( M \) in SMV’s description language. Thus, it suffices to code the property we want to verify using the specification language of SMV resulting in a CTL (computation tree logic) formula \( p \), and run SMV with inputs \( M \) and \( p \).

### 3 Musical Examples

**Example 1.** Consider the example from (12) in which different rhythm patterns are defined. Let us define a process *Metronome* which ticks every \( t \) time units:

```java
Metronome is
while(true) {
    check(e);
    tick;
    read(t);
    write(t:=t+t)
}
```

where the execution of event \( e \) is restricted by the constraint \( wait(e, \#t) \) defined as
\[
wait(X, Y) \leftarrow \#Y < X, wait(X+, \#Y).
\]

This is, event \( e \) is allowed to be executed only after \( t \) time units have elapsed, event \( e+ \) is executed after \( 2t \) time units have elapsed, \( e++ \) after \( 3t \) time units have elapsed, etc. This process could interact with an accelerator process

```java
Accelerator is
while(true) {
    read(signal);
    read(t);
    if (signal=1 and t>k) then write(t:=t-k)
}
```

which accelerates the ticks of *Metronome* by a value of \( k \) if some other process execute \( write(signal=1) \). This other process, the director process, could be defined as
Director is
    check(f);
    write(signal=1);
    check(g);
    write(signal=0)

and restricting the execution of events f and g by the constraint

\[
\#start < f \land \#end < g.
\]

In general, the director process may act as a conductor signaling the start and ending of a variety of events using different signal variables. For instance, here the accelerator and the director process share the signal variable which results in a tempo acceleration within the start-end time interval.

A refinement of the director process may be to require that it accelerates the tempo every b units of time (within the start-end time interval), in which case a mechanism similar to the one in the metronome process can be implemented.

Director is
    check(f);
    write(signal=1);
    while(t1+start < end) {
        check(h);
        read(t1);
        if (t1-start) mod b = 0 then
            write(signal=1)
        else
            write(signal=0);
    }
    write(t1:=t1+t1);
    check(g);
    write(signal=0)

The execution of event h is restricted by the constraint \(\text{wait}(h, \#t1)\) defined as before. Events f and g are restricted as before.

The metronome process can be straightforwardly adapted to implement a process which plays a melody by replacing the code of \texttt{tick}. It is easy to imagine a director process signaling \texttt{accelerandos} and \texttt{ritardandos} resulting in tempo variations in the melody.

Example 2. Consider a music system in which there are two processes: a leader process which plays a (possibly improvised) melody, and the follower process which reproduces the melody (up to transposition) with some delay. The leader and the follower play notes in the sets \(\{N_1, N_2, \ldots, N_k\}\) and \(\{T_1, T_2, \ldots, T_k\}\), respectively. A note transposition function \(f(N_i) = T_1, \ldots, f(N_k) = T_k\) gives for each leader’s note the follower’s note to be played later. Additional constraints state that (a) the follower cannot play in advance to the leader, i.e. the follower has to wait until the leader plays a note in order to reproduce the transposed note, and (b) the leader cannot play more than \(n\) notes in advance to the follower, i.e. if the leader plays \(n\) notes none of which have been transposed by the follower, then the leader is forced to wait for at least one follower’s note before continue playing the next note. The two processes shown below model this problem.

Leader is
    while (true) {
        get_next_note(X);
        check(a);
        play_note(X);
        check(b);
    }
    play_transposed(Y);
    check(d)

Follower is
    while (true) {
        check(c);
        listen(X);
        Y = f(X);
        play_transposed(Y);
    }

The property that follower never attempts to play in advance can be expressed by the constraint \(p(b, c)\) defined as

\[
p(X, Y) \leftarrow X < Y, p(X+, Y+).
\]

and the property that the leader never plays more than \(n\) notes in advance can be specified by the constraint \(p(d, a++n)\) defined as above. For instance, restricting a leader from playing more than 2 notes in advance can be specified by the constraint \(p(d, a++2)\). The code for \texttt{play} \_ \texttt{note}(X), in addition to playing the note, adds the note to a buffer. The code for \texttt{listen}(X) removes a note from the buffer.

Using these constraints, i.e. \(p(b, c)\) and \(p(d, a++2)\), it is straightforward to model the leader-follower system as a transition system, to translate the transition system into a model checker description language, and automatically apply model checking to verify properties about the system using SMV (for details see (11)).

Example 3. Consider a situation where we have a network of audio processing modules which are linked to each other and must respect a set of temporal constraints (e.g. some processing tasks have to be done before others). We may assign an event to each processing module and specify the execution order of the system as a set of Tempo constraints over these events. Figure 1 shows a network of processes and their links as well as the Tempo constraints that enforce the processes’ execution order. Note that the system constraints \(e_1 < e_2, e_2 < e_3, e_3 < e_4, e_3 < e_5, e_4 < e_6, e_5 < e_6\) specify a partial order among events \(e_i (1 \leq i \leq 6)\) and that the execution order of events \(e_4\) and \(e_5\) is left undefined (i.e. the execution order of \(e_4\) and \(e_5\) is not deterministic). As illustrated by previous examples, it is straightforward to state similar constraints for iterative processes: it suffices to replace \(e_i < e_j\) by \(p(e_i, e_j)\) where \(p(X, Y)\) is defined as in example 2. Thus, the execution order of a more realistic system in which the audio processing modules are constantly receiving and processing data is specified by the constraints
Note that it is transparent to the user if the processes are distributed or executing in the same machine.

Figure 1: Audio processes execution order specified by the constraints: $e_1 < e_2, e_2 < e_3, e_3 < e_4, e_3 < e_5, e_4 < e_6, e_5 < e_6$.

4 Conclusion

In this paper we have described work in progress on modeling musical processes using a logic-based concurrent programming language. The model provides a declarative formalism in which (a) the synchronization issues are treated as orthogonal to the system base functionality, and (b) distribution of audio processes in different machines is transparent to the user. The model allows the specification of complex interactions among musical processes. The model provides an executable specification for musical problems and provides a formal framework to reason about them. An important point of the work reported here is that it is possible to formally prove musical properties of processes. A significant subset of Tempo programs, which includes the musical examples in this paper, can be compiled into finite state systems. Finite state systems are amenable to automatic verification using model checking techniques since their observable behavior can be finitely represented. We plan to apply our formalism to music composition and real-time music performance.

Acknowledgments: This work is supported by the Spanish TIC project ProMusic (TIC 2003-07776-C02-01).

References


