APOPCALEAPS: Automatic Music Generation with CHRiSM

Jon Sneyers Danny De Schreye

K.U.Leuven, Belgium

Abstract

We present a new system for automatic music generation, in which music is modeled using very high level probabilistic rules. The probabilistic parameters can (at least in principle) be learned automatically from examples, resulting in a system for personalized music generation.

1 Introduction

Algorithmic music composition is an intriguing subfield of artificial intelligence. Typical examples include the works of David Cope [3] and Iannis Xenakis [14], and the elaborate Band-in-a-Box software [6].

Pearce et al. [8] have identified four main motivations for developing computer programs which compose music. In their terminology, this paper describes the design of a tool for composers which can also be used for the computational modelling of musical styles. As Conklin pointed out [2], a general statistical model can be applied to both synthesis (automatic composition) and analysis (modelling of styles, e.g. classification). Indeed, a statistical analytic model can (in principle) be sampled to generate music.

Probabilistic logic programming [5] is an extension of logic programming [1] which allows programmers to express both statistical and relational knowledge in a natural way. The probabilistic logic language PRISM [9] has been shown to subsume many well-known statistic models like stochastic context-free grammars and hidden Markov models. In [13], PRISM was used to implement a music model which was used for both synthesis and analysis. Recently, a new and higher-level rule-based probabilistic logic programming language has been proposed, called CHRiSM [11]. In this paper we present a music generation and learning system, called APOPCALEAPS, implemented in CHRiSM.

The structure of this paper is as follows. First we give a brief overview of APOPCALEAPS in Section 2. In Section 3 we introduce CHRiSM. Section 4 discusses some program excerpts. We conclude in Section 5.

2 The APOPCALEAPS System

Figure 2 gives a schematic overview of APOPCALEAPS (an acronym for "Automatic POP Composer And LEArner of ParameterS"). We briefly give an overview of the components of the system.

A graphical user interface provides a front-end to the underlying CHRiSM program. This interface essentially allows the user to tweak an input query for the CHRiSM program, specifying some desired properties of the generated music. The default query is shown in Fig. 1. This query has the following meaning. We want a piece with four voices: melody, bass, chords and drums. The shortest possible note for the melody and drums is set to a 16th note, while for the bass and chords it is set to an 8th note. Names of MIDI instruments to be used to render the voices are given. The range of the melody is set to the interval of 5 semitones below central C to 16 semitones above central C. The biggest interval between two

```
voice(melody), shortest_duration(melody,16),
voice(bass), shortest_duration(bass,8),
voice(chords), shortest_duration(chords,8),
voice(drums), shortest_duration(drums,16),
instrument(melody,'soprano sax'),
instrument(bass,'electric bass (pick)'),
instrument(chords,'electric guitar (jazz)'),
set_range(melody,c,4,-5,16), max_jump(melody,5),
set_range(bass,c,3,-17,5), max_jump(bass,17),
chord_style(offbeat), max_repeat(melody,2),
key(major), meter(2,4), tempo(120), measures(8)
```

Figure 1: The default query for APOPCALEAPS.

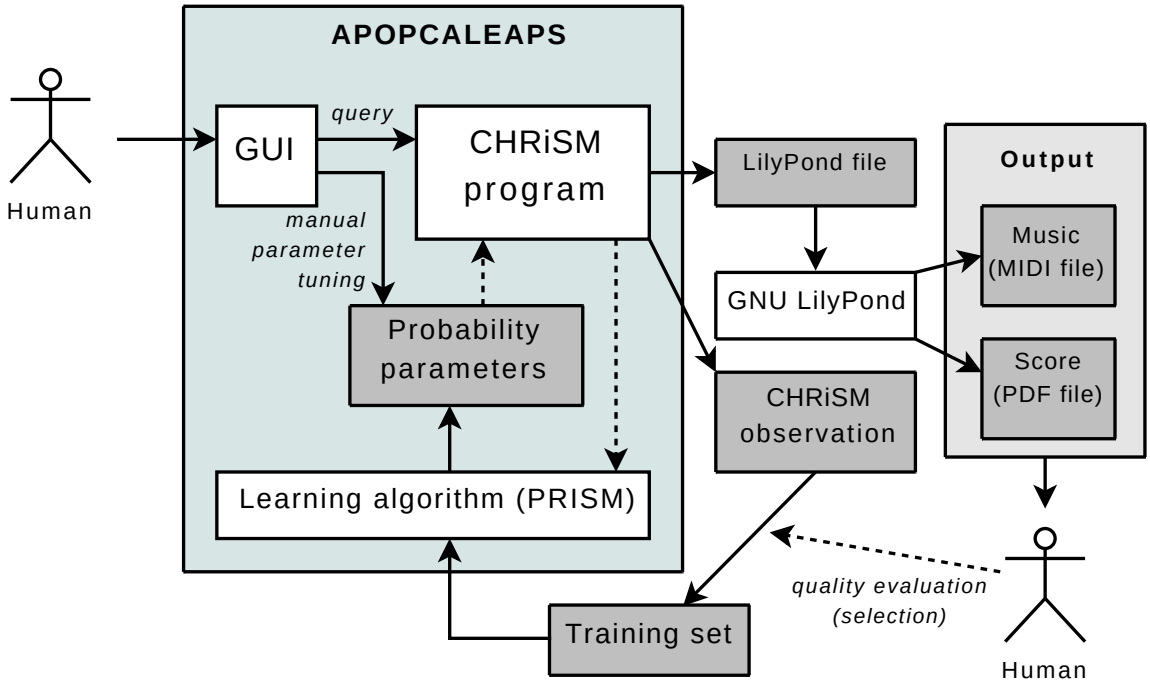

Figure 2: An overview of the APOPCALEAPS system.

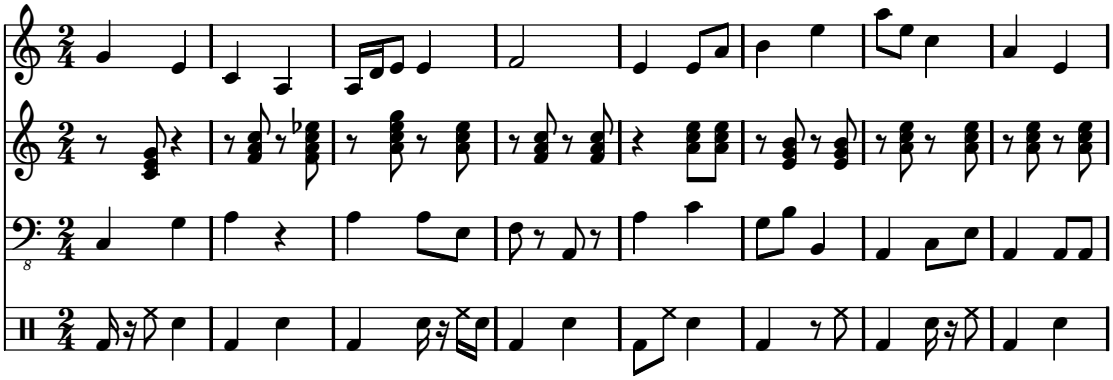

Figure 3: Example output of the APOPCALEAPS system (fragment).

consecutive melody notes is set to 5 semitones. The bass has a lower range and is allowed to make bigger jumps. Chords are preferably on off-beats, and the melody should not have more than two consecutive repeated notes. The piece should be in a major key. The meter is 2/4, the tempo is 120 bpm, and the length of the piece to be generated is 8 measures.

The GUI also allows users to modify the probability distributions used inside the CHRiSM program. Based on the query and these probability parameters, the CHRiSM program generates output, rendered by LilyPond [7] as both a score and a MIDI file. Figure 3 shows a fragment of an output example.

Next, the user listens to the generated music and selects the good pieces (according to his own taste). The selected pieces are used as a training set for a learning algorithm that adjusts the probability parameters. Using the new parameters, more (and hopefully better) output is generated and selected for addition to the training set. This iterative interactive process results in a personalized music generation system.

Although all major parts of the APOPCALEAPS system have been implemented, we will focus on the CHRiSM program that produces the output, given a query. Learning works in principle, but it is computationally too expensive to be tested in practice on nontrivial examples. The main issue is that the current CHRiSM system is not yet able to deal efficiently with large output spaces — although this is the subject of ongoing work [10]. Hence we leave this part of the APOPCALEAPS system as future work, and use

manually tuned probability parameters for now.

The above description of a personalized music generation system, as illustrated in Fig. 2, is only one way to use APOPCALEAPS. It could also be used for other purposes. For example, classification of existing music could be done by training several instances of the model, one per category, based on a number of manually classified examples. Unknown examples can then be classified by computing their likelihood under each model instance. Other potential applications include finding the most likely chord sequence underlying a given piece, completing partial pieces, generating variations on a piece, etc.

The core component of the APOPCALEAPS system is a CHRiSM program. In Section 4 we will look at excerpts from this program. But first we introduce CHRiSM.

3 CHRiSM

CHRiSM (CHance Rules induce Statistical Models) is a new programming language for probabilistic logic learning [11]. It is based on a combination of CHR and PRISM.

CHR (Constraint Handling Rules) [4] is a high-level language extension based on multi-headed rules. Originally CHR was designed as a special-purpose language to implement constraint solvers, but in recent years it has matured into a general purpose programming language [12]. Being a language *extension*, CHR is implemented on top of an existing programming language, which is called the *host language*. An implementation of CHR in host language X is called CHR(X). For instance, several CHR(Prolog) systems are available. The implementation of CHRiSM is based on a CHR(PRISM) system.

PRISM (PRogramming In Statistical Modeling) [9] is a probabilistic extension of Prolog. It supports several probabilistic inference tasks, including sampling, probability computation, and expectation-maximization learning. We assume the reader to be familiar with Prolog [1].

3.1 Syntax and Semantics

A CHRiSM program $\mathcal{P}$ consists of a sequence of *chance rules*. Chance rules rewrite a multiset $\mathbb{S}$ of data elements, which are called (CHRiSM) *constraints* (mostly for historical reasons). Syntactically, a constraint c(X$_1$,..,X$_n$) looks like a Prolog predicate: it has a functor c of some arity n and arguments X$_1$,..,X$_n$ which are Prolog terms. The multiset $\mathbb{S}$ of constraints is called the *constraint store* or just *store*. The initial store is called the *query* or *goal*, the final store (after exhaustive rule application) is called the *answer* or *result*.

Chance rules. A chance rule is of the following form: P ?? Hk \ Hr <=> G | B.
where P is a probability expression (as defined below), Hk is a conjunction of (kept head) constraints, Hr is a conjunction of (removed head) constraints, G is a guard condition (a PRISM goal to be satisfied), and B is the body of the rule. If Hk is empty, the rule is called a *simplification* rule and the backslash is omitted; if Hr is empty, the rule is called a *propagation* rule and it is written as "P ?? Hk ==> G | B". If both Hk and Hr are non-empty, the rule is called a *simpagation* rule. The guard G is optional; if it is removed, the "|" is also removed. The body B is a conjunction of CHRiSM constraints, PRISM goals, and probabilistic disjunctions (as defined below). A non-probabilistic chance rule has P equal to 1; the "1 ??" may be removed and it corresponds to a regular CHR rule.

Intuitively, the meaning of a chance rule is as follows: If the constraint store $\mathbb{S}$ contains elements that match the head of the rule and furthermore, the guard G is satisfied, then we can consider rule application. The subset of $\mathbb{S}$ that matches the head of the rule is called a rule *instance*. Depending on the probability expression P, the rule instance is either ignored or it actually leads to a rule application. Every rule instance may only be considered once. Rule application has the following effects: the constraints matching Hr are removed from the constraint store (the constraints matching Hk are kept), and then the body B is executed, i.e. PRISM goals are called and CHRiSM constraints are added to the store.

Probability expressions. A probability expression P is either a number or arithmetic expression that indicates a probability, or it is an *experiment name*. The latter is a Prolog term which should be ground when the rule is considered. It indicates an unknown probability distribution. Initially, unknown probabilities are set to a uniform distribution. They can be changed manually using PRISM's set_sw/2 builtin, or automatically using PRISM's EM-learning algorithm. The arguments of an experiment name can include *conditions*, which are of the form "cond C". Such arguments are evaluated at runtime and replaced by either "yes" or "no", depending on whether call(C) succeeded or failed.

Probabilistic disjunction. The body B of a CHRiSM rule is defined as a conjunction of PRISM goals, CHRiSM constraints, and explicit or implicit probabilistic disjunctions of bodies. Explicit probabilistic disjunctions are of the form "D1:P1 ; ... ; Dn:Pn", where a disjunct Di is chosen with probability Pi. The probabilities should sum to 1. Implicit probabilistic disjunctions are of the form "P ?? D1 ; ... ; Dn", where P is an experiment name determining the probability distribution.

PRISM predicates. Since CHRiSM is based on CHR(PRISM), all features of PRISM can also be used in the body of chance rules. In particular, the PRISM built-in msw(E,V) can be used to randomly sample experiment E and unify V with its outcome value. The outcome space of experiment E has to be declared using values(E,VL), where VL is a (finite) list of ground Prolog terms. The probability distribution can be set using set_sw(E,PL), where PL is a list of probabilities (one for every outcome value).

Operational Semantics. The abstract operational semantics of a CHRiSM program $\mathcal{P}$ is given by a (probabilistic) state-transition system that resembles the abstract operational semantics ω_t of CHR [12]. We use the symbol $\omega_t^{??}$ to refer to the abstract operational semantics of CHRiSM and $\xrightarrow[\mathcal{P}]{p}$ to denote a transition with probability p. It is formally defined in [11]. If all rule probabilities are 1 and the program contains no probabilistic disjunctions — i.e. if the CHRiSM program is actually a CHR program — then the $\omega_t^{??}$ semantics boils down to the ω_t semantics of CHR.

3.2 Full and Partial Observations

A full observation Q <==> A denotes that there exist a series of probabilistic choices such that a derivation starting with query Q results in the answer A. A partial observation Q ===> A denotes that an answer for query Q contains at least A: Q ===> A holds if Q <==> B with A $\trianglelefteq$ B.

The following PRISM built-ins can be used to query a CHRiSM program:

- sample Q : probabilistically execute the query Q;
- prob Q <==> A : compute the probability that Q <==> A holds, i.e. the chance that the choices are such that query Q results in answer A;
- prob Q ===> A : compute the probability that an answer for Q contains A;
- learn(L) : perform EM-learning based on a list L of observations

For more information about CHRiSM we refer to [11].

3.3 Examples

To illustrate the semantics of CHRiSM rules, we give a few examples. As a first toy example, consider the following CHRiSM program for tossing a coin:

```
toss <=> head:0.5 ; tail:0.5.
```

The query `toss` results in `head` or `tail`, with 50% chance each. The query `toss,toss` has four possible outcomes, each with 25% chance: "`head,head`", "`head,tail`", "`tail,head`", and "`tail,tail`".

3.3.1 Rock-paper-scissors

Consider the following CHRiSM program simulating "rock-paper-scissors" players:

```
player(P) <=> choice(P) ?? rock(P) ; scissors(P) ; paper(P).
rock(P1), scissors(P2) ==> winner(P1).
scissors(P1), paper(P2) ==> winner(P1).
paper(P1), rock(P2) ==> winner(P1).
```

We assume that each player has his own fixed probability distribution for choosing between rock, scissors, and paper. This is denoted by using `choice(P)` as the probability expression for the choice in the first rule: the probability distribution depends on the value of `P` and thus every player has his own distribution. However, these distributions are not known to us. By default, the unknown probability distributions for, say, `tom` and `jon` are therefore both set to the uniform distribution (cf. Figure 4). Here is a possible interaction:

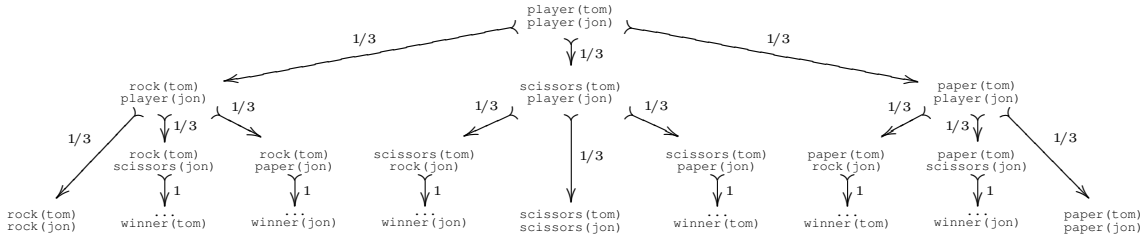

Figure 4: A derivation tree for the rock-paper-scissors example.

```
?- sample player(tom),player(jon)
player(tom),player(jon) <==> rock(jon),rock(tom).
?- sample player(tom),player(jon)
player(tom),player(jon) <==> rock(jon),paper(tom),winner(tom).
?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.333333
```

Now suppose that we watch 100 games, and want to use our observations to obtain a better model of the playing style of both players. If we can fully observe these games, then this is easy: we can just use the frequency with which each player played rock, paper or scissors as an estimate for the probability of him making that particular move. The situation becomes more difficult, however, if the games are only partly observable. For instance, suppose that we do not know which moves the players made, but are only told the final scores: `tom` won 50 games, `jon` won 20, and 30 games were a tie. Deriving estimates for the probabilities of individual moves from this information is less straightforward. PRISM comes with a built-in implementation of the EM-algorithm for performing parameter estimation in the presence of missing information [9]. We can use this algorithm to find plausible corresponding distributions:

```
| ?- learn([ (50 times  player(tom),player(jon) ===> winner(tom)),
             (20 times  player(tom),player(jon) ===> winner(jon)),
(30 times  player(tom),player(jon) ===> ~winner(tom),~winner(jon))])
```

The PRISM built-in `show_sw` shows the learned probability distributions, which do indeed (approximately) lead to the observation frequencies, e.g.:

```
| ?- show_sw
Switch choice(jon): 1 (p: 0.60057) 2 (p: 0.06536) 3 (p: 0.33406)
Switch choice(tom): 1 (p: 0.08420) 2 (p: 0.20973) 3 (p: 0.70605)
| ?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.499604
```

3.3.2 Random graphs

Suppose we want to generate a random directed graph, given its nodes. The following chance rule generates every possible directed edge with probability 50%:

```
0.5 ?? node(A), node(B) ==> edge(A,B).
```

Given the query `node(x), node(y), node(z)`, there are 6 rule instances to be considered for this rule: {`A=x,B=y`}, {`A=x,B=z`}, {`A=y,B=x`}, {`A=y,B=z`}, {`A=z,B=x`}, {`A=z,B=y`} . If for example only the first two rule instances are actually applied, then the result consists of the three original nodes plus `edge(x,y)` and `edge(x,z)`. In total there are 2^6 possible results for this query, each with probability $(1/2)^6$.

4 The APOPCALEAPS system in detail

The CHRiSM program at the core of the APOPCALEAPS system consists of about 50 CHRiSM rules (about 150 lines of code). Besides the actual program, there is some auxiliary code (about 100 lines of code) and the code to write out the output in LilyPond syntax (about 150 lines of code).

The program uses 7 parametrized probabilistic experiments, which give rise to 92 probability distributions in total. Fig. 5 shows the names and outcome spaces of these experiments. As will become clear, we have made several simplifying assumptions.

```
values(chord_choice(C), [c,g,f,am,em,dm]).
values(note_choice(V,C,B), [c,d,e,f,g,a,b,r]).
values(octave_choice(mid), [-2,-1,0,+1,+2]).
values(octave_choice(low), [0,+1,+2]).
values(octave_choice(high), [-2,-1,0]).
values(drum_choice(B), [bd,sn,hh,cymc,r]).
values(chord_type(_,B), [0,7,r]).
values(split_beat(V), [no,yes]).
values(join_notes(V,_,_), [no,yes]).
```

Figure 5: Parametrized probabilistic choices.

The first experiment, chord_choice, determines the chord sequence. We restrict the set of possible chords to C major, G major, F major, A minor, E minor and D minor. This covers a large subset of pop music (modulo transposition). The experiment is parametrized by the previous chord. Section 4.1 explains how this experiment is used.

The second experiment, note_choice, determines the pitch of the notes of the melody and bass. For simplicity, we only allow pitches in the scale of C major (no accidentals). Besides these 7 pitches, rest (r) is also part of the domain. The experiment is parametrized by the type of voice (currently only melody or bass), the current chord, and the current abstract beat position. Abstract beat positions are defined as follows: "first" for the first beat of a measure, "strong" for other on-beats of a measure, "weak" for the off-beats of a measure, "prestrong" for positions just before an on-beat, and "weakest" for all other positions. We abstract the rhythm in this way to simplify the model.

The third experiment, octave_choice, determines the octave of the notes. Jumps of up to two octaves up or down are supported. If the current octave is near the border of the voice range, the domain is reduced.

The fourth experiment, drum_choice, determines the notes of the drum. The choices are restricted to bass drum, snare, hi-hat, crash cymbal, and rest. The experiment is parametrized by abstract beat.

The fifth experiment, chord_type, determines the chord voice. This voice always plays a variant of the current chord, as determined by chord_choice. The only options are: the chord itself (a triad), a seventh chord, and rest. This experiment is parametrized by the chord style (part of the query) and by abstract beat.

The sixth and seventh experiments determine the rhythm; we will discuss this later, in Section 4.2. Because of space restrictions, it is impossible to explain the entire CHRiSM program, even though it is not that long. Instead we will zoom in on a few rules.

4.1 Chord Sequence Generation

To keep things simple we assign one chord per measure and we model the chord sequence as a simple Markov chain as shown in Fig. 6 (arrow widths indicate probabilities). The transition probabilities were manually set to these values based on personal taste and experience. As mentioned in Section 2, they can, in principle, also be learned automatically from examples. Fig. 7 shows the CHRiSM rules to implement the chord sequence generation. We now briefly discuss them.

The input query specifies the key (key/1) and the number of measures (measures/1). The output chord sequence is encoded as mchord(X,C) constraints, where C is the chord for measure number X.

The first four rules make sure that if a piece is in a major key, the first and last measure get the chord "C major", while if it is in a minor key, they get "A minor".

The next two rules generate a sequence of measure/1 constraints (one for every measure), along with a sequence of next_measure/2 constraints that connect every measure with its successor.

The last rule does the actual chord sequence generation. It reads as follows: if the chord of measure X is C, and the measure following X is Y (where Y is not the last measure M), then we choose a new chord NextC based on the probabilistic experiment chord_choice(C), and set the chord of measure Y to be NextC.

4.2 Rhythm Generation

The rhythm is generated in three steps. Firstly, for every voice, beat/5 constraints are generated for every beat in every measure, according to the meter specified in the query. The meaning of the arguments is

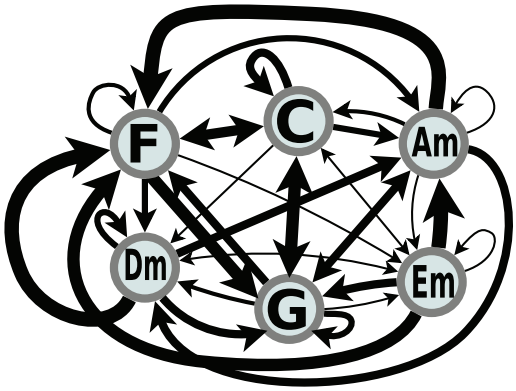

Figure 6: Chord transitions.

```
key(major), measure(1) ==> mchord(1,c).
key(major), measures(N) ==> mchord(N,c).
key(minor), measure(1) ==> mchord(1,am).
key(minor), measures(N) ==> mchord(N,am).

measures(N) ==> make_measures(N).
make_measures(N) <=> N>0 | measure(N),
                     N1 is N-1, next_measure(N1,N),
                     make_measures(N1).

mchord(X,C), next_measure(X,Y), measures(M)
      ==> Y < M | msw(chord_choice(C),NextC),
                  mchord(Y,NextC).
```

Figure 7: CHRiSM rules for chord transitions.

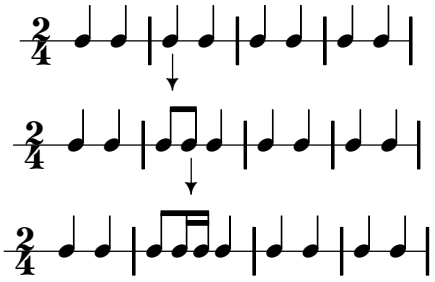

Figure 8: Beat splitting.

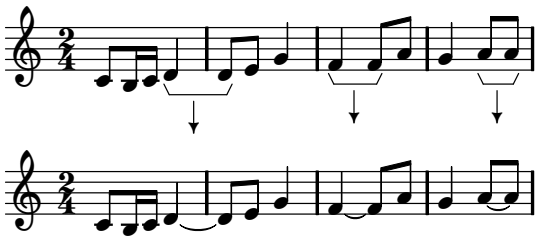

Figure 9: Beat joining.

as follows: beat(V,M,N,X,D) represents an event for voice V, in measure number M, at beat number N (=0,1,2,...), and sub-beat position X (a real number $0 \leq X < 1$), with the duration of a Dth note. Also, next_beat/7 constraints are generated to link beats to their successor.

Secondly, some of the beats are "split in two", as illustrated in Fig. 8. Beat splitting is handled by a (recursive) chance rule; for every voice, splitting can only continue as long as the beats have a duration that is longer than the shortest duration declared in the input query. For simplicity, the probability of splitting only depends on the voice — of course this could be refined to also take into account the current depth and/or abstract beat position and other relevant parameters.

Thirdly, some of the beats are joined again by a tie, if two consecutive notes have the same pitch. This step is only applied after the pitches have been generated. Fig. 9 illustrates this process. Beat joining is handled by another chance rule, whose application probability depends on three parameters: the voice, and two boolean conditions that indicate the degree of syncopation inherent in the tie. The first condition is true if the consecutive notes are in the same measure, the second condition is true if both notes belong to the same beat. The three examples in Fig. 9 correspond to the cases (no,no), (yes,no), and (yes,yes). Fig. 10 lists the two chance rules that generate the rhythm.

5 Conclusion and Future Work

We have presented the APOPCALEAPS system for generating and learning pop music based on chance rules, implemented in CHRiSM. This is one of the first applications of CHRiSM, and it illustrates the power and expressiveness of chance rules. Several existing approaches for music generation and analysis can be combined and generalized by formulating them in CHRiSM. For example, (hidden) Markov models are easily expressed as chance rules. Therefore, in our opinion, CHRiSM is an excellent programming language for automatic composition in the sense of [8].

The APOPCALEAPS system allows parameter learning from previously generated pieces, or pieces from any source, as long as they can be represented in our internal notation (this may require extending the outcome spaces of the probabilistic experiments of Fig. 5). In order to make learning computationally

```
split_beat(V) ??
    meter(_,OD), shortest_duration(V,SD)  \  beat(V,M,N,X,D), next_beat(V,M,N,X,Mn,Nn,Z)
      <=> D<SD |    D2 is D*2, Y is X+1/(D2/OD),
                    next_beat(V,M,N,X,M,N,Y), next_beat(V,M,N,Y,Mn,Nn,Z),
                    beat(V,M,N,X,D2), beat(V,M,N,Y,D2).

join_notes(V,cond Ma=Mb,cond Na=Nb) ??
    next_beat(V,Ma,Na,Xa,Mb,Nb,Xb), note(V,Mb,Nb,Xb,SameNote)
     \  note(V,Ma,Na,Xa,SameNote)  <=> note(V,Ma,Na,Xa,SameNote+' ~').
```

Figure 10: CHRiSM rules for splitting and joining notes.

feasible, the underlying CHRiSM system has to be sufficiently efficient. This is the subject of ongoing work [10]. An efficient CHRiSM system will enable us to evaluate the APOPCALEAPS system by testing its accuracy for genre classification.

Since the probability distributions are currently tuned manually, the number of parameters was kept rather small. If learning is sufficiently efficient, the number of parameters can be increased by refining the rules to take more factors into account or by adding arbitrary hidden states.

The APOPCALEAPS system and example output music can be downloaded at the following website:

http://people.cs.kuleuven.be/jon.sneyers/apopcaleaps/

References

[1] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[2] Darell Conklin. Music generation from statistical models. In *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*, pages 30–35, 2003.

[3] David Cope. *Computer Models of Musical Creativity*. MIT Press, 2005.

[4] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[5] Lise Getoor and Ben Taskar, editors. *Statistical Relational Learning*. MIT Press, 2007.

[6] PG Music Inc. Band-in-a-box. http://www.band-in-a-box.com/, 2010.

[7] Han-Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, Firenze, Italy, May 2003.

[8] Marcus Pearce, David Meredith, and Geraint Wiggins. Motivations and methodologies for automation of the compositional process. *Musicae Scientiae*, 6(2), 2002.

[9] Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, 2008.

[10] Jon Sneyers. Result-directed CHR execution. In Peter Van Weert and Leslie De Koninck, editors, *CHR 2010*, 2010.

[11] Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. In M. Hermenegildo, I. Niemelä, and T. Schaub, editors, *26th International Conference on Logic Programming*, Edinburgh, UK, July 2010.

[12] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming*, 10(1), January 2010.

[13] Jon Sneyers, Joost Vennekens, and Danny De Schreye. Probabilistic-logical modeling of music. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL'06)*, pages 60–72, Charleston, SC, USA, January 2006.

[14] Iannis Xenakis. *Formalized Music: Thoughts and Mathematics in Music*. Pendragon Press, 1992.