APOPCALEAPS: Automatic Music Generation with CHRiSM

Jon Sneyers and Danny De Schreye
K.U.Leuven, Belgium
1. **CHRiSM**
   - Constraint Handling Rules (CHR)
   - CHRiSM
   - Syntax & semantics of CHRiSM
   - PRISM features in CHRiSM

2. **APOPCALEAPS**
   - Overview
   - Constraint declarations
   - Chord generation
   - Rhythm generation
   - Note generation

3. **Conclusion**
1 CHriSM
   - Constraint Handling Rules (CHR)
   - CHriSM
   - Syntax & semantics of CHriSM
   - PRISM features in CHriSM

2 APOPCALEAPS

3 Conclusion
High-level language \textit{extension}

- different host languages (originally and mostly Prolog)
- e.g. CHR(Prolog), CHR(Haskell), CHR(Java), CHR(C)

Multi-headed committed-choice guarded rewrite rules

Originally: designed for writing constraint solvers

Today: general-purpose programming language
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint </2.
reflexivity @ X ≤ X <=> true.
antisymmetry @ X ≤ Y, Y ≤ X <=> X = Y.
idempotence @ X ≤ Y \ X ≤ Y <=> true.
transitivity @ X ≤ Y, Y ≤ Z ==> X ≤ Z.
```

Example execution:

Goal: A ≤ B, B ≤ C, C ≤ A

Store:
Example 1: less-or-equal solver

Typical “solver” CHR program:

:— chr_constraint <=/2.

reflexivity @ X≤X <=> true.
antisymmetry @ X≤Y, Y≤X <=> X=Y.
idempotence @ X≤Y \ X≤Y <=> true.
transitivity @ X≤Y, Y≤Z ==> X≤Z.

Example execution:

Goal: A≤B, B≤C, C≤A
Store: A≤B
Example 1: less-or-equal solver

Typical “solver” CHR program:

```chrism
:- chr_constraint <=/2.
reflexivity @ X≤X <=> true.
antisymmetry @ X≤Y, Y≤X <=> X=Y.
idempotence @ X≤Y \ X≤Y <=> true.
transitivity @ X≤Y, Y≤Z ==> X≤Z.
```

Example execution:

- Goal: A≤B, B≤C, C≤A
- Store: A≤B, B≤C
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity  @ X = X <=> true.
antisymmetry @ X = Y, Y = X <=> X = Y.
idempotence   @ X = Y \ X = Y <=> true.
transitivity  @ X = Y, Y = Z =/> X = Z.
```

Example execution:

- Goal: A ≤ B, B ≤ C, C ≤ A
- Store: A ≤ B, B ≤ C
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X <= X <- true.
antisymmetry @ X <= Y, Y <= X <- X = Y.
idempotence @ X <= Y \ X <= Y <- true.
transitivity @ X <= Y, Y <= Z => X <= Z.
```

Example execution:

- Goal: A <= B, B <= C, C <= A
- Store: A <= B, B <= C, A <= C
Example 1: less-or-equal solver

Typical “solver” CHR program:

```
:- chr_constraint <=/2.
reflexivity @ X<=X <=> true.
antisymmetry @ X<=Y, Y<=X <=> X=Y.
idempotence @ X<=Y \ X<=Y <=> true.
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

Example execution:

- Goal: A<=B, B<=C, C<=A
- Store: A<=B, B<=C, A<=C, C<=A

Jon Sneyers and Danny De Schreye
Automatic Music Generation using CHRiSM
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X<=X <=> true.
antisymmetry @ X<=Y, Y<=X <=> X=Y.
idempotence @ X<=Y \ X<=Y <=> true.
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

Example execution:

- Goal: A≤B, B≤C, C≤A
- Store: A≤B, B≤C, A≤C, C≤A
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.

reflexivity @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: A <= B, B <= C, C <= A
- Store: A <= B, B <= C, A = C
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X =< X <=> true.
atomsymmetry @ X =< Y, Y =< X <=> X = Y.
idempotence @ X =< Y \ X =< Y <=> true.
transitivity @ X =< Y, Y =< Z => X =< Z.
```

Example execution:

- Goal: A =< B, B =< C, C =< A
- Store: A =< B, B =< A, A = C
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity  @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence  @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: \( A \leq B, \ B \leq C, \ C \leq A \)
- Store: \( A \leq B, \ B \leq A, \ A = C \)
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: $A \leq B$, $B \leq C$, $C \leq A$
- Store: $A = B$, $A = C$
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity  @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence  @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: A <= B, B <= C, C <= A
- Store: A = B, A = C

← answer
Example 2: Prime number generation

- Typical “general-purpose” CHR program:

```prolog
:- chr_constraint upto/1, prime/1.
upto(1) <=> true.
upto(N) <=> N>1 | prime(N), upto(N-1).

prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

- First two rules implement a loop, generating a sequence

```prolog
prime(2), ..., prime(n)
```

- Third rule filters out the non-prime numbers:
  - if I divides J, then J is not a prime
  - the rule removes such non-primes
Probabilistic CHR: CHRIISM [ICLP 2010]

based on CHR(PRISM)

PRISM: PRogramming In Statistical Modeling
[Sato 1995, Sato & Kameya 1997]

CHRIISM: CHance Rules induce Statistical Models
PRISM built-in \texttt{msw/2} can be used in CHR(PRISM) programs

- \texttt{msw(+Experiment,-Result)}: Experiment is ground at runtime; Result gets a random value based on a predefined discrete probability distribution

For example:

\begin{verbatim}
values(coin, [head, tail]).
:- set_sw(coin, [0.5, 0.5]).
\end{verbatim}

\begin{verbatim}
toss <= msw(coin, X), write(result=X).
\end{verbatim}

CHRiSM is “syntactic sugar” for CHR(PRISM)

- relatively simple kind-of-source-to-source transformation
Chance rules (may) have two kinds of probabilities:
- Rule: application probability
- Body: probabilistic disjunction

**Syntax: rule with probability Prob**

```
Prob ?? Head <=> Guard | Body.
```

Normal CHR rules: “1 ??”

**Syntax: probabilistic disjunction (in rule body)**

- Fixed probability distribution: (cf. CP-Logic [Vennekens et al. 2006])
  
  ```
  D1:Prob1 ; D2:Prob2 ; ... ; DN:ProbN
  ```

- Unknown/learnable probability distribution:
  
  ```
  Prob ?? D1 ; D2 ; ... ; DN
  ```
Chance rules (may) have two kinds of probabilities:
- Rule: application probability
- Body: probabilistic disjunction

Syntax: rule with probability Prob

Prob ?? Head <-> Guard | Body.

normal CHR rules: “1 ??”

Syntax: probabilistic disjunction (in rule body)

fixed probability distribution: (cf. CP-Logic [Vennekens et al. 2006])

D1:Prob1 ; D2:Prob2 ; ... ; DN:ProbN

unknown/learnable probability distribution:

Prob ?? D1 ; D2 ; ... ; DN
Syntax of CH RiSM

- Chance rules (may) have two kinds of probabilities:
  - Rule: application probability
  - Body: probabilistic disjunction

Syntax: rule with probability Prob

\[
\text{Prob} \ ?\ ? \ \text{Head} \iff \text{Guard} \mid \text{Body}.
\]

Normal CHR rules: “1 ??”

Syntax: probabilistic disjunction (in rule body)

Fixed probability distribution: (cf. CP-Logic [Vennekens et al. 2006])

\[
D1:\text{Prob1} \ ; \ D2:\text{Prob2} \ ; \ldots \ ; \ DN:\text{ProbN}
\]

Unknown/learnable probability distribution:

\[
\text{Prob} \ ?\ ? \ D1 \ ; \ D2 \ ; \ldots \ ; \ DN
\]
Operational semantics as usual \((\omega_t, \omega_r, \omega_p)\)

Two differences:

- rule application can be skipped (with probability \(1 - P\))
- probabilistic disjunctions in the body: one disjunct is randomly chosen (committed-choice)
Features of PRISM

- PRISM has many nice features, a.o.:
  - Probabilistic execution (sample)
  - Probability computation (prob)
  - EM-learning (learn)

- These features can also be used in CHRiSM
Probabilistic execution: sample goal
- starting from goal, apply CHriSM rules (just like in CHR)
- rules with probability $P$ are skipped with probability $1 - P$
- in a probabilistic disjunction, exactly one disjunct is chosen

Probability computation: prob goal $\iff$ result
- compute probability that “sample goal” gives “result”
- prob goal $\implies$ result
  compute probability that “sample goal” gives something of the form “result, otherstuff”

EM-learning: learn(observations)
- observations: a list of observations of the form “goal $\iff$ result” or “goal $\implies$ result”
- compute an assignment to the unknown probabilities such that the likelihood of the observations is maximized
1. CHRiSM

2. APOPCALEAPS
   - Overview
   - Constraint declarations
   - Chord generation
   - Rhythm generation
   - Note generation

3. Conclusion
Goal: automatically generate original, royalty-free music

Could be used in:
- railway stations, airports, waiting rooms, stores (places where people are used to listing to crappy music)
- computer games (music style influenced by game events)
- tools for human composers, e.g. to get inspiration

Should have a (probabilistic) learning component:
- some musical rules are known, most are not
- for some genres (e.g. Renaissance counterpoint), exhaustive enumeration of the rules is possible, but still:
  - need an expert who knows all rules
  - need to write out all rules in some formalism (tedious!)
  - from all pieces that satisfy the rules, some will be better than others; how to pick a solution?
- *de gustibus non disputandum est* : the system has to be able to adjust to the musical taste of the user
Overview

Constraint declarations
Chord generation
Rhythm generation
Note generation

Human

APopCALeaPs

GUI

CHRiSM program

Query

Probability parameters

Manual parameter tuning

Learning algorithm (PRISM)

Training set

Output

Music (MIDI file)

Score (PDF file)

GNU LilyPond

Quality evaluation (selection)

Human

Jon Sneyers and Danny De Schreye Automatic Music Generation using CHRiSM
[demo of the APopCALeaPs system]
APopCALeaPs generation process

APopCALeaPs

GUI

query

manual parameter tuning

Probability parameters

CHRiSM program

output (LilyPond file)
APopCALeaPs generation process

GUI

query

APopCALeaPs

input constraints:
key, #measures,
#voices, ...

chord generation

rhythm generation

Probability parameters

CHRiSM program

write output in LilyPond format

output constraints

output (LilyPond file)

manual parameter tuning

Jon Sneyers and Danny De Schreye
Automatic Music Generation using CHRiSM
% inputs
:- chrism measures(+int), meter(+int,+duration), repeats(+int),
   key(+key), shortest_duration(+voice,+duration), tempo(+int),
   voice(+voice), range(+voice,+note,+int,+note,+int),
   max_jump(+voice,+int), instrument(+voice,+),
   chord_style(+cstyle), max_repeat(+voice,+int).

:- chr_type key ---> major ; minor.
:- chr_type voice ---> melody ; chords ; bass ; drums.
:- chr_type note ---> c ; d ; e ; f ; g ; a ; b.
:- chr_type duration ---> 2 ; 4 ; 8 ; 16 ; 32.
:- chr_type cstyle ---> offbeat ; long ; onbeat.
% outputs
:- chrism  measure(+measure), mchord(+int,+chord),
        beat(+voice,+measure,+int,+float,+duration),
        note(+voice,+measure,+int,+float,+),
        octave(+voice,+measure,+int,+float,+).

:- chr_type chord ---> c ; d ; e ; f ; g ; a ; b ;
                 cm ; dm ; em ; fm ; gm ; am ; bm.
:- chr_type measure == int.
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).
Chord generation (2)

% simple Markov chain chord progression
mchord(A,Chord), next_measure(A,B), measures(M)  
  ==> B < M |  
      msw(chord_choice(Chord),NextChord),  
      mchord(B,NextChord).

Jon Sneyers and Danny De Schreye
Automatic Music Generation using CHRiSM
Rhythm generation (1)

\[ \frac{3}{4} \quad \boxed{\text{\textbullet \textbullet \textbullet \textbullet \textbullet \textbullet \textbullet \textbullet \textbullet}} \]

% create one beat per beat
meter(N,D), voice(V), measure(M) ==> make_beats(N,D,M,V).
make_beats(0,_,_,_,_) <=> true.
make_beats(N,D,M,V) <=> N > 0 |
N1 is N-1, next_beat(V,M,N1,0,M,N,0),
beat(V,M,N1,0,D), make_beats(N1,D,M,V).

meter(N,D), next_measure(M,M2)
\backslash next_beat(V,A,B,C,M,N,E) <=> next_beat(V,A,B,C,M2,0,0).
% split some of the beats in two
split_beat(V) ??
meter(_,OD), measures(LastM), phase(split), shortest_duration(V,SD)
\ beat(V,M,N,X,D), next_beat(V,M,N,X,NM,NN,NX) <=< D<SD, M <= LastM |
   D2 is D*2, X2 is X+1/(D2/OD),
next_beat(V,M,N,X,M,N,X2), next_beat(V,M,N,X2,NM,NN,NX),
beat(V,M,N,X,D2), beat(V,M,N,X2,D2).
% choose first note
make_notes_measure(1), beat(V,1,0,0,D), mchord(1,C) =>
  V \== drums, V \== chords |
  abstract_beat(1,0,0,AB),
  soft_msw(note_choice(V,C,AB),Note),
  note(V,1,0,0,Note).

% choose next note and octave
make_notes_measure(M), beat(V,M,N,X,D), mchord(M,C),
octave(V,M1,N1,X1,00), next_beat(V,M1,N1,X1,M,N,X) =>
  V \== drums, V \== chords |
  abstract_beat(M,N,X,AB),
  soft_msw(note_choice(V,C,AB),Note),
  note(V,M,N,X,Note),
  ( Note == r \-> octave_d(V,M,N,X,0)
  ; find_octave_d(V,M,N,X,00) ).
Why soft_msw?

- normal msw randomly picks a value and commits to it
- soft_msw picks a value, if it fails, it picks a different value
- useful construct to combine probabilistic choice and integrity constraints

% check max_jump constraint - fail (and backtrack) if it is violated
max_jump(V,MInt), octave(V,M1,N1,X1,00), note(V,M1,N1,X1,ON),
note(V,M,N,X,NN), next_beat(V,M1,N1,X1,M,N,X) \ octave(V,M,N,X,NO) <=>
  interval(ON,00,NN,NO,Int), Int > MInt | fail.

% check max_repeat constraint - fail (and backtrack) if it is violated
max_repeat(V,N), same_note_counter(V,A,B,C,N) <=> fail.
\% two successive notes of the same pitch can be joined

\begin{verbatim}
\text{join_notes}(V, \text{cond } M=M2, \text{cond } N=N2) \equiv \text{phase(join_notes}(M)), \text{note}(V, M, N, X, \text{Note}), \\
\text{next_beat}(V, M, N, X, M2, N2, X2), \text{note}(V, M2, N2, X2, \text{Note}) \implies V \\|= \text{drums} \| \text{tied}(V, M, N, X).
\end{verbatim}
1. CHRiSM

2. APOPCALEAPS

3. Conclusion
CHRiSM allows very high-level music modelling
  - the entire music generation program is less than 500 lines (including whitespace and comments), most of which is output formatting etc.

CHRiSM is an interesting hybrid programming paradigm
  - Usual approach in music generation: either probabilistic or constraint-based
  - First time a combined approach is tried (AFAIK)
... so there is a lot of future work:

- Current model of music is very simplistic
  - add (more) hidden states etc.
  - collaborate with a musicologist to improve it?
  - trade-off with learning efficiency
- Music analysis / automatic classification
  - train several instances of the model with different genres/composers/styles/...
  - probability of a piece in each model indicates likelihood of belonging to a genre/...
- Experimental evaluation
  - can do this for analysis (but requires a lot of tedious data preparation)
  - how to do this for synthesis? Turing test?
Questions?