A NEW APPROACH FOR CONSTRAINT PROGRAMMING IN MUSIC USING RELATION DOMAINS

Sascha Van Cauwelaert, Gustavo Gutiérrez, Peter Van Roy

Université catholique de Louvain
ICTEAM Institute, Louvain-la-Neuve, Belgium
sascha.vancauwelaert@gmail.com
gustavo.ggutierrez@gmail.com
peter.vanroy@uclouvain.be

ABSTRACT

Constraint programming (CP) has been used for several decades in music composition and analysis. It has served as the underlying technology of different tools that allow composers to compute with musical abstractions (e.g., notes, scores). However, the traditional domains used in musical CP, namely finite domains (integers) and finite sets (integer sets), are not well suited to represent and express properties on structured information such as a score in a compact and efficient way. This paper introduces a new domain for musical CP, namely relations, where a relation is a set of integer n-tuples. It proposes new constraints on relations and shows how to use them for musical composition. A single relation variable can represent a score of any size and any transformation between scores. The result is a system that directly supports computing with musical abstractions at a high abstraction level more pleasant to composers. The relation domain and its constraints are implemented using Binary Decision Diagrams and are provided as a library in the Gecode platform.

Keywords : Constraint programming, relation decision variables, music, computer-aided composition, binary decision diagrams, Gecode, OpenMusic, relational algebra.

1. INTRODUCTION

Constraint programming (CP) is a common approach to tackle problems in music ([5, 3]). To avoid musicians the effort of going to the low level of using integer and set decision variables directly, modeling frameworks such as *Strasheela* [4] have emerged. These frameworks provide musical abstractions for modeling the problems and then translate that model into an equivalent one that can be tackled by a constraint solver.

During the translation process the framework expresses the problem as a CSP[1]. That is, in terms of decision variables, their possible values, and constraints on them. The efficiency of the solution process depends on the kind of variables and constraints supported by the solver, usually integer and integer set decision variables. This paper presents *relation decision variables*, a new domain in which the represented information consists of relations rather than just integers or sets. The proposed domain brings several advantages. First, it provides a new level of abstraction for modeling which can lead to succinct and more expressive models. Second, it offers a new set of constraints that can be used to express high level properties on the variables. And third, it provides a new solver that works directly at this high level, thus making it possible to achieve higher efficiency.

1.1. Motivating example for relation domains

Consider the problem of finding a *round* (also known as *simple canon*), with a *leader voice* V_0 and one *follower voice* V_1. V_1 has to be played o onsets after V_0. Additionally, when played together some given harmony rules must be respected. The value o is called the offset between the onsets of V_1 and V_0.

The solution of this problem is a score. From the problem, we need to consider at least the *pitch* and *onset* parameters for every note in it. We represent by R_S the resulting score. R_S is a binary relation (e.g. set of pairs) in which every element has the form $\langle pitch, onset \rangle$. The same representation is used for V_0 and V_1.

The offset is represented by a unary relation *Offset*. Any element in this relation will represent the offset performed by a voice. So, for the particular case of one follower voice, *Offset* contains only one element. For the harmony let us assume an input binary relation *Consonant* with elements of the form $\langle pitch_i, pitch_j \rangle$. This relation groups pitches that are accepted to be heard at the same time[2].

Expression (1) states the round property between the two voices and (2) ensures that both voices get in the score we need to find. Expression (3) links pitches of R_S that are played at the same onset. Those links are then forced to belong to *PP*. The harmony constraint (4) is ensured by

[1] Constraint programming terminology for a *Constraint Satisfaction Problem.*

[2] Therefore, the harmony constraint will be basic and simplified.

stating that all the pitch links are consonant.

$$(V_0 \times \mathit{Offset}) \underset{2}{\smile} \mathit{Plus} = V_1 \quad (1)$$

$$R_s = V_0 \cup V_1 \quad (2)$$

$$R_s \underset{1}{\smile} R_s = PP \quad (3)$$

$$PP \subseteq \mathit{Consonant} \quad (4)$$

The way of expressing the offset between the voices can look intimidating at first sight. By doing $V_0 \times \mathit{Offset}$ we compute a ternary relation with elements of the form $\langle \mathit{pitch}, \mathit{onset}, \mathit{offset} \rangle$ that basically computes any note in V_0 concatenated with any offset. By using the operation $\underset{2}{\smile}$ (introduced further in Section 3) we compute new notes of the form $\langle \mathit{pitch,onset'} \rangle$ where *onset'* is the new onset at which the note has to be played. Expression (1) contains the relation *Plus*. This is a ternary relation in which every element $\langle x, y, z \rangle$ has the property $x + y = z$. In order to link pitches played at a same offset, we use the operation $\underset{1}{\smile}$ on R_S with itself, but with permuted components. That is, we *compose* (see Section 3) tuples of the form $\langle \mathit{pitch}, \mathit{onset} \rangle$ with tuples of the form $\langle \mathit{onset}, \mathit{pitch} \rangle$.

Some strengths of using the introduced domain can be appreciated in this model. The constraints are stated in a high level way and over natural music concepts like scores. A score as a set of notes is represented directly as a single relation variable. The information used to represent each note is kept together as a tuple inside a score. To show these strengths clearly, we provide a second model of the same problem using integer and integer sets which are traditional domains in CP.

1.2. Traditional approach using integer domains

We now model the same problem using only decision variables over integers and sets. To model a score V we use two arrays[3], one that represents the pitches (5) and the other one the onsets (6). The arrays must be big enough to store all the notes in the largest possible score we consider. The notes themselves are referenced by the indices of the arrays. A note n in the voice V is then represented by its pitch $\mathit{Pitches}_V(n)$ and its onset $\mathit{Onsets}_V(n)$. As there is no sense in having the same note (i.e. same pitch and onset) inside a given voice we have to add the constraint (7).

$$\mathit{Pitches}_V = \langle p_1, \ldots, p_l \rangle \quad (5)$$

$$\mathit{Onsets}_V = \langle o_1, \ldots, o_l \rangle \quad (6)$$

$$\forall_{i,j \in \{1,\ldots,l\}} : i \neq j \wedge \mathit{Pitches}_V(i) = \mathit{Pitches}_V(j) \implies \mathit{Onsets}_V(i) \neq \mathit{Onsets}_V(j) \quad (7)$$

We represent the two voices of the problem by the arrays $\mathit{Pitches}_{V0}$, Onsets_{V0}, $\mathit{Pitches}_{V1}$, and Onsets_{V1}. Expressions (8) and (9) state the round property between the scores.

$$\forall_{i \in \{1,\ldots,l\}} : \mathit{Onsets}_{V1}(i) = \mathit{Onsets}_{V0}(i) + o \quad (8)$$

$$\forall_{i \in \{1,\ldots,l\}} : \mathit{Pitches}_{V1}(i) = \mathit{Pitches}_{V0}(i) \quad (9)$$

To enforce the harmony constraint, we use an array of sets *Consonant*. A pitch p_i is consonant with a pitch p_j if $p_j \in \mathit{Consonant}(i)$. This is stated by expression (10). Notice that similar constraints must be imposed for pitches of notes of a given voice to be consonant.

$$\forall_{i,j \in \{1,\ldots,l\}} : \mathit{Onsets}_{V0}(i) = \mathit{Onsets}_{V1}(j) \implies \mathit{Pitches}_{V0}(i) \in \mathit{Consonant}(\mathit{Pitches}_{V1}(j)) \quad (10)$$

1.3. Comparison of the two approaches

Let us compare the relation-oriented modeling of the problem with the integer-oriented approach. The relational model is simpler and it keeps together the notions of pitch and onset that characterize a note. The model works for scores of any length with no ad hoc size parameters. On the other hand, the integer model uses two arrays of ad hoc size and a constraint to represent the notion of a note. Moreover, that constraint is translated to $l \times (l-1)$ binary constraints on the solver. The properties represented by (8) and (9) are translated into l constraints each. The disadvantage of this translation is that it makes constraint inference limited to two properties of the same note at a time while ignoring useful information when the score itself is considered. This reduces the efficiency of the CP solver in the integer model.

There are several additional advantages to use relation domains. First, it can be seen that generalising the problem to n voices is easily done in the first model but not in the second one. The only thing to do is to allow *Offset* to contain several elements. Notice that in that case V_1 will actually contain several follower voices. Moreover, if *Offset* is not fixed at modeling time, the number of follower voices are not fixed either. *Offset* could also be constrained in some way, e.g. all offsets are multiple of a given value, for instance 4. Even more, as it will be shown in Subsection 5.4, it is possible to offset several musical parameters of a score, and then to constrain that general "offset structure". If we were working with integer and sets, those generalisations would not be straightforward at all (e.g. we have to use one array per parameter per voice and use a special value for silences).

1.4. Contributions

This paper defines the relation constraint domain and explores ways to use it in the field of music. The notions of relation and tuple allow to represent musical entities such as score and note in a more concise and natural way than traditional domains. General purpose constraints provide the building blocks for representing new abstractions that benefit final users such as composers. These constraints complement other domains by allowing constraint inference at higher levels of abstraction.

[3] Actually we need an array per every feature of the score we want to represent.

The constraint domain is implemented on top of the *Gecode* [12] constraint library, its sources are freely available [9] and an interface for integrating it with OpenMusic [1] is in development[4].

1.5. Document structure

Section 2 shows how relations can express key concepts in music. Section 3 introduces CP with relations and defines the new constraints of this domain. Section 4 explains how CP with relations can be used to model musical CSPs. Section 5 gives modeling examples that target music composition and Section 6 shows how to solve a larger example. Section 7 briefly presents our implementation in Gecode. Conclusions and future work are presented in Section 8.

2. MUSICAL CONCEPTS AS RELATIONS

Current music constraint systems[5] all share the same problem *internally*[6], highlighted by the composer J. Kretz in [10] : a lack of structure in the music representation that forces to handle musical parameters independently, making them difficult to interact. J. Kretz proposed then to use an (hypothetical) "organized structure for notes (a bundle of information containing many parameters like pitch, duration, [...])" in order to define "more 'intelligent' rules" [10]. In a sense, this is what we are proposing in this work. This section defines the concepts needed to link this general idea with constraint programming on relations.

We define the two concepts of *Musical Bundle* (MB) and *Musical Bundle Sets* (MBS) as a way of representing musical concepts in terms of tuples and relations, which are the basic concepts of our new constraint system.

Musical bundle. *Is a set of pairs where each pair combines a musical parameter with an integer value.* An example of a MB is a note defined by several musical parameters, such as the following tuple:

$$tuple = \langle pitch, duration, onset, instrument, \ldots \rangle \quad (11)$$

Musical bundle set. *Is a set of musical bundles.* Examples of MBSs are a chord, a bar, or a score. But an MBS can express more abstract musical concepts as well, for example a transformation between two scores can also be represented as an MBS. When modeling musical problems, we will use the terms MBS and relation variable interchangeably.

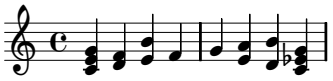


Figure 1. Score represented by a MBS (relation). Every note is represented by a MB (tuple).

[4]https://github.com/svancauw/GeLiSo

[5]Term introduced in [3, 4].

[6]Some provide abstractions to hide this problem to the user.

In figure 1, the note C played on the first beat and the note F played on the second beat can be respectively represented by the tuple $\langle 60,1 \rangle$ and the tuple $\langle 65,2 \rangle$ (if we use MIDI values for the pitch parameter). The score represented by figure 1 is then represented by the set of all the tuples (i.e., a relation) that represents the different notes played on the score.

3. CONSTRAINTS ON RELATIONS

Constraint programming on relations is about using *relation decision variables* along with constraints that enforce properties on these variables. It also provides search abstractions and predefined generic heuristic strategies for solving models with this kind of variables.

A relation decision variable (relation variable for short) represents a relation out of a set of possible relations. A relation is a set of tuples of the same arity. A tuple is an element of the cartesian product of the finite set of integers[7]: $\mathscr{U} = \{x : 0 \leq x \leq k\}$. The arity of a tuple is the number of elements that belongs to it. For instance, tuples of arity n are elements of the set: $\underbrace{\mathscr{U} \times \ldots \times \mathscr{U}}_{n \text{ times}}$. This set is also represented as $\mathscr{U}^n$.

The domain of a relation variable X, denoted D_X, is the set of possible relations that X can be assigned to. A constraint C on a set of relation variables $\{X_1, \ldots, X_n\}$ will ensure that the domain of every variable X_i only contains relations that satisfy it. This is also called constraint inference. A variable is considered determined or assigned when its domain contains one and only one relation.

The constraints that can be used on relation variables come from two fields: set theory and relational algebra [7]. As a relation variable represents a relation it does make sense to express properties like: $X \cap Y = Z$ and $X = \bar{Y}$; where X, Y and Z are relation variables. Properties from the relational algebra include for instance: $X \underset{C}{\bowtie} Y = Z$, which states that the join of the two relations X and Y on the components in C must be the relation Z.

The notion of tuple concatenation is used for the definitions of the constraints. Given two tuples $r = \langle r_1, \ldots, r_n \rangle$ and $s = \langle s_1, \ldots, s_m \rangle$, we represent its concatenation by $r{+}{+}s = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$. The arity of the resulting tuple $|r{+}{+}s| = n + m$.

Projection. This is a binary constraint on variables X and Y of different arities. It takes a constant i as an extra parameter, the number of components on the right of X that are projected. Its semantics is:

$$\prod_i X = Y \equiv \forall t_1, \ldots, t_{|X|} : \quad (12)$$
$$\langle t_1, \ldots, t_{|X|-(i-1)}, \ldots, t_{|X|} \rangle \in X \iff$$
$$\langle t_{|X|-(i-1)}, \ldots, t_{|X|} \rangle \in Y$$

As a special case, this constraint creates a channel between a relation and a set decision variable, if $i = 1$.

[7]A fixed and large enough k ensures that $\mathscr{U}$ is finite.

Join. This is a ternary constraint on relations. It takes also a constant i, which represents the number of components on the right of X and on the left of Y that are considered by the constraint. Its semantics is:

$$X \underset{i}{\bowtie} Y = Z \equiv \forall r,s : \exists u : |u| = i \wedge \quad (13)$$
$$((r{+}{+}u \in X \wedge\ u{+}{+}s \in Y) \iff (r{+}{+}u){+}{+}s \in Z)$$

Compose. This is a special form of the join constraint where the joined components are removed from the result. It matches the semantics of relation composition from the relational algebra [7]. Its semantics is:

$$X \underset{i}{\smile} Y = Z \equiv \forall r,s : \exists u : |u| = i \wedge \quad (14)$$
$$((r{+}{+}u \in X \wedge\ u{+}{+}s \in Y) \iff r{+}{+}s \in Z)$$

Intuitively, a compose collects all paths that combine a step in X and a step in Y, to form a step in Z. Join does the same but additionally exposes the intermediate steps in Z.

Confluent join. This is a special case of *join* with an additional confluence condition. The additional condition ensures that it collects only those combined steps in Z for which all possible first steps in X can always find a next step in Y that continues to the same result.

$$X \overset{\forall}{\underset{i}{\bowtie}} Y = Z \equiv \forall r,s : \forall u : |u| = i \wedge \quad (15)$$
$$(((r{+}{+}u \in X \implies u{+}{+}s \in Y) \wedge$$
$$(\exists t : |t| = i \wedge r{+}{+}t \in X \wedge\ t{+}{+}s \in Y)) \iff$$
$$(r{+}{+}u){+}{+}s \in Z)$$

Confluent compose. This is a special case of *compose* with an additional confluence condition.

$$X \overset{\forall}{\underset{i}{\smile}} Y = Z \equiv \forall r,s : \forall u : |u| = i \wedge \quad (16)$$
$$(((r{+}{+}u \in X \implies u{+}{+}s \in Y) \wedge$$
$$(\exists t : |t| = i \wedge r{+}{+}t \in X \wedge\ t{+}{+}s \in Y)) \iff$$
$$r{+}{+}s \in Z)$$

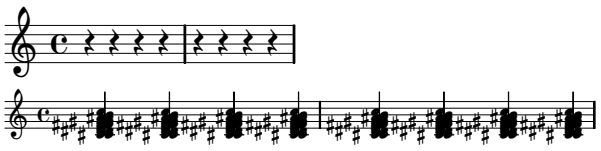
Permutation. This allows to impose that a relation is equal to another one but with some components permuted. Using it, we can apply *projection*, *join*, *compose*, *confluent join* and *confluent compose* constraints on components of relations without regarding their position. We do not define it here because we will not use it explicitly in the presented models, in order to keep them more readable.

4. MODELING WITH MUSICAL BUNDLE SETS

This section presents some of the new possibilities that CP on relations offers in the musical field. When we define a new musical problem in terms of relation variables, we need first to declare these variables. We declare them by giving the minimum and maximum MBS that the variable can assume. We then impose constraints on these relation variables. The constraint solver then uses a combination of propagation and heuristic search to find values of the relation variables that satisfy the constraints. For example, suppose that we are interested in finding a musical piece M with some specific characteristics. The piece itself is represented by the relation variable M with the following minimum and maximum MBS:

$$M \in [\emptyset \ldots \{\{60,\ldots,72\} \times \{0,\ldots,7\}\}] \quad (17)$$

This notation says that the minimum MBS of M is empty (i.e. 8 beats of silence) and the maximum MBS of M is the musical piece that contains all the notes from middle C to C one octave higher, occurring on the first 8 beats [8]. The minimum and maximum MBS of M are represented by the following scores :



Additionally we want our score to respect some composition rules. This is done by imposing constraints such as those given in Section 5. In some sense, this corresponds to what a composer does when he wants to compose a new musical piece. He starts from nothing and can potentially add every note. From this provided "musical material", he decides what he will use to construct his piece by imposing some "properties" on it.

4.1. MBS as a transformation of MBSs

An MBS can be used to define a transformation between two other MBSs. For instance, each note $\langle pitch, onset\rangle$ in the original MBS can be transformed into $\langle pitch', onset'\rangle$ in the new MBS. How the new values for the attributes are computed depends on an MBS that represents the transformation itself. As a particular example, let us consider a score *score* like the one used in the introduction and a relation T to represent the intended musical transformation. Every element of T has the form:

$$\langle pitch_{start}, onset_{start}, pitch_{end}, onset_{end}\rangle \quad (18)$$

To obtain the transformed score (represented by the binary relation $score_{trans}$) resulting from the transformation of *score* by T we impose the compose constraint

$$score_{trans} = score \underset{2}{\smile} T \quad (19)$$

In this way we ensure that any MB of *score* that can be transformed by T has one (or several) respective(s) transformed MB in $score_{trans}$.

More generally, it is actually possible to link any two relation variables with a third relation variable that represents the transformation, if we code the transformation in terms of integers. All three relation variables in this link will participate in the solution process. That is, the CP solver can use the third variable to calculate the transformed relation, but it can also use the two relation variables to calculate the third variable.

[8] We assume here a signature $4/4$ and quarter notes. For this example we only consider *pitch* and *onset* as the relevant parameters of the MBs.

4.2. MBS as an aggregation of MBSs

One way of linking a set of MBSs together is to aggregate them into another MBS. This can be done easily by adding a parameter to each MBS and assigning it a different value for each MBS, i.e. all the MBs of a given MBS have the same value for this new parameter. Formally, if S is the set of MBSs to aggregate, we have

$$\forall\, s_i \in S,\ \forall\, mb \in s_i : mb{+}{+}i \in A \tag{20}$$

where A is the MBS that aggregates the set of MBSs S, and $mb{+}{+}i$ represents an MB constructed from the MB mb in which a new parameter with value i has been added. All this can be achieved using CP on relations. If R_A and R_{s_i} are relations that represent respectively A and the i^{th} MBS of S, we have

$$R_A = \bigcup_i R_{s_i} \underset{0}{\bowtie} \{\langle i \rangle\} \tag{21}$$

In this way, we can for instance aggregate two scores (e.g., the left and right hand parts of a piano score) to form a new score. We can then constrain the resulting score in some way, and this will be reflected in the parts of the aggregation.

5. EXAMPLES OF MUSICAL CONSTRAINTS

This section presents several examples to show how to use relation constraints to solve musical problems. Notice that, as said in Section 3, we will ignore here the use of permutation constraints, in order to clarify the text. Also, we will use an MBS named *score* that contains MBs of the form $\langle pitch, onset, duration \rangle$ to represent a score and an MBS named *Chords* in which every MB has the form $\langle pitch, ChordIndex \rangle$ in order to represent a set of chords. In each MB, *ChordIndex* is an index identifying a chord (i.e. a set of pitches) and *pitch* is one of the pitches of the chord indexed by *ChordIndex*. By chord, we mean here any set of pitches (e.g. C major, beginning from middle C, in its fundamental position).

5.1. Forbid simultaneous chords

We propose here a constraint that is hard to express without relation variables. The constraint is the following : in a given score, we want to forbid to **hear** chords of some sets of complete chords altogether, even if the different notes constituting the chords do not begin to be played at the same time. The constraint takes three musical parameters into account : *pitch*, *onset* and *duration*. We need an MBS *FSChordSets* that will be the set of forbidden-simultaneous-chords sets. Every MB in *FSChordSets* has the form $\langle SetIndex, ChordIndex \rangle$, where *ChordIndex* is an index identifying a chord and *SetIndex* is an index identifying a set of chords that cannot **all** be heard completely[9] simultaneously. The first thing to do to apply the constraint is to transform the original score *score* into a score for which all the notes with a duration n become n consecutive notes of duration 1. To do this, we need the transformation MBS represented by the relation

$$T_{\text{duration}} = \{\langle X, Y, Z \rangle : Z \in [X, X+Y-1] \wedge Y > 0\} \tag{22}$$

To get the transformed score (represented by the relation variable $score_{\text{dur}=1}$), we only apply the constraint

$$score_{\text{dur}=1} = score \underset{2}{\smile} T_{\text{duration}} \tag{23}$$

After that, we need to get all the complete chords *heard* at a given time in *score*, that is, all the complete chords *played* at a given onset in $score_{dur=1}$. This can be done using the constraint :

$$IO = Chords \overset{\forall}{\underset{1}{\smile}} score_{\text{dur}=1} \tag{24}$$

where IO is a binary relation linking a given onset with chords completely played at that onset. The last part needed to express the constraint is to impose that on a given onset (of $score_{\text{dur}=1}$), we cannot have all the chord indexes of a given chord set index.

$$\emptyset = FSChordSets \overset{\forall}{\underset{1}{\smile}} IO \tag{25}$$

Notice that if the constraint is not respected, we know because of which chords, by using the operation *confluent compose*. This information can help to lead the search.

5.2. Score harmonization

Another constraint that can be easily expressed is the harmonization of a score **during the search** to determine that score. So, this information can be used to lead the search in some way. One more time, we use here several musical parameters together in order to express what we want. Here, we will only use a part of a score (for instance, a bar), and look for one chord harmonizing that part. Moreover, in some cases, we can propose several chord alternatives. The first step to express the constraint is to get all the notes of the given part and to keep the expected constitutive notes of the chord. In our case, we simply keep the notes with a minimal duration value. To get the relevant notes, we create a new MBS (represented by the relation variable *score'*) that contains only those notes. To do this, we impose the constraint

$$score' = score \underset{2}{\bowtie} (Onsets \times Duration_{min}) \tag{26}$$

where *Onsets* and $Duration_{min}$ are respectively the set (unary relation) of onsets of the part[10] and the set (unary relation) of all possible durations, with a minimum lower bound. After that, we just need to identify the chord indexes of chords that can be used to harmonize the given part of the score. To do this, we use the constraint

$$harmScore = score' \overset{\forall}{\underset{1}{\smile}} Chords \tag{27}$$

[9] I.e. all the notes of the chord are heard.

[10] Generally this set does not contain "holes" but it could.

where *harmScore* is a relation linking onsets (and possibly other musical parameters different from the pitch) and chord indexes. The harmonization is then done.
The interesting thing with this constraint is that it is possible to use partial information inferred from that constraint. For instance, if at some time during the search we know that a majority of chords of a given tonality has been used to harmonize the score, we will first try to impose the use of pitches of that tonality in the score.

5.3. Simplified orchestration

Suppose we want to distribute a piece of music *score* on two instruments i_1 and i_2 (it could be generalized to n instruments easily). Scores of these instruments can also be represented by two relation variables $score_{i_1}$ and $score_{i_2}$, whose components have the same semantics as those of *score*. The only thing we have to impose are :

$$score = score_{i_1} \cup score_{i_2} \tag{28}$$

and

$$score_{i_1} \cap score_{i_2} = \emptyset \tag{29}$$

There are of course lots of solutions to this problem. But it becomes interesting when we begin to add other constraints on $score_{i_1}$ and $score_{i_2}$, for instance one is the transformation of the other by some transform relation.

5.4. Score made of musical patterns

We present here a constraint that is in some way an extension of a constraint used for Michael Jarrell's "Congruences" (presented by Serge Lemouton in [11]). The constraint that will be expressed here states that a score is made only of a given set of musical patterns, i.e. subscores in which notes are represented by several parameters. Let this set be represented by the relation *Patterns*. Every tuple of *Patterns* has the form

$$\langle indexPattern, pitch, onset, duration, \ldots, score_{param_i}, \ldots, score_{param_n} \rangle$$

Patterns is a relation of arity $n+1$ to be used to create a relation that represents a score with n parameters. The component *indexPattern* is used to identify one pattern in the set of patterns.

We would like to impose that a given score is made from some patterns of this set of patterns, and nothing else. In order to do this, we will simply get the used patterns from the current score (found during the search), and recreate a new score from those used patterns. If the recreated score is equal to the initial one, only patterns of *Patterns* have been used, that is, the constraint is satisfied[11].

[11] Of course at least one pattern must be used or the score is empty, which is a trivial solution for this constraint.

The possibility to shift the patterns in some components should exist. To allow this, we use a relation variable *Shift* that will represent all the shifts applied to patterns. Every tuple of *Shift* has the form

$$\langle indexPattern, offset_{param_1}, \ldots, offset_{param_i}, \ldots, offset_{param_n} \rangle$$

Using *Shift*, we can get a relation variable that represents the set of all possible shifted patterns. We explain in the following how, but we must first introduce the relation $Plus^{3.n}$:

$$t \in Plus^{3.n} \iff \forall i \in [0, \ldots, n-1] : t(i) + t(i+n) = t(i+2n) \tag{30}$$

$Plus^{3.n}$ can be defined in terms of *Plus* (defined in Section 1):

$$Plus^{3.n} = Permute_{i \leftrightarrow (i\%n)*3+i/n}(\underbrace{Plus \times \cdots \times Plus}_{n}) \tag{31}$$

where $Permute_{i \leftrightarrow (i\%n)*3+i/n}$ is a function that permutes all the components i of a relation with the component $(i\%n)*3+i/n$, respectively. Using the relation $Plus^{3.n}$, we can work with a variable set of shifted patterns

$$Patterns_{\text{shifted}} = Patterns \underset{1}{\bowtie} Shift \underset{2.n}{\smile} Plus^{3.n} \tag{32}$$

With $Patterns_{\text{shifted}}$, we are able to retrieve all the patterns used in the score *score*, even if they have been shifted in the score.

$$usedPatternsInScore = score \overset{\forall}{\underset{n}{\bowtie}} Patterns_{\text{shifted}} \tag{33}$$

Having used *confluent join* instead of *confluent compose* allows us to keep the information about the score. The relation variable *usedPatternsInScore* gives us the information about how and which patterns have been used in *score*. The only thing that remains to be done is to check if the used (shifted) patterns construct exactly the score *score*. We only need to remove the pattern indexes from *usedPatternsInScore* and check (impose) equality with *score* :

$$score = \prod_{n} usedPatternsInScore \tag{34}$$

Adding some constraints on *Shift* can help to have more control on the solution. For instance, we can avoid to have the exact same shifts for different patterns. Moreover, we can notice that this constraint can be used incrementally. Indeed, we can first use some patterns in a given search, and afterwards use patterns that contain some of those patterns as subpatterns, and so on. Eventually, we are here more general than specified because the definition of *confluent join* contains only an implication and not a double implication : this allows to use only parts of the patterns, i.e. subpatterns. In order to force the use of complete patterns, we simply need to modify the definition of *confluent join* with a double implication instead of the implication.

6. A COMPLETE EXAMPLE

This section presents a practical problem solved in the OpenMusic environment using constraint programming on relations. To work with Gecode inside OpenMusic we use GeLiSo ("Gecode in common Lisp using Sockets"). The musical CSP is the one presented in Section 1 with some additional constraints:

$$V_0 \cap V_1 = \emptyset \tag{35}$$

$$V_{0_{part1}} = V_0 \underset{1}{\bowtie} Onsets_{part1} \tag{36}$$

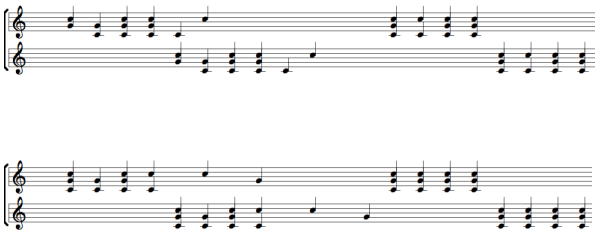
$$V_{0_{part2}} = V_0 \underset{1}{\bowtie} Onsets_{part2} \tag{37}$$

$$(V_{0_{part1}} \times OffsetPart) \underset{2}{\smile} Plus = V_{0_{shifted}} \tag{38}$$

$$V_{0_{shifted}} \cap V_{0_{part2}} = \emptyset \tag{39}$$

Expression (35) states that the two voices do not have any note in common. Expressions (36) and (37) allow to get some note subsets of the voice V_0. With expression (38), we are able to shift the MBS $V_{0_{part1}}$ on 8 onsets later. Eventually, thanks to expression (39), we impose that the shifted part of V_0 does not have any common note with $V_{0_{part2}}$.

Here are two solutions to the musical constraint satisfaction problem presented above. In both cases, we only allow to use the MIDI pitch values 60, 67 and 72 and they can all be heard together. In the first case, we impose that the parts of V_0 that cannot be the same are the first 4 beats and the beats from 8 to 12. In the second case, the constraint is applied on the first 5 beats and on the beats from 8 to 13. Experimentally, we noticed that from 6 beats, no solution can be found in a reasonable time (in this case less than 2 minutes). But no special heuristic is used for now.



7. IMPLEMENTATION

We now explain how the new relation constraints are implemented. The implementation is available as an extension of the Gecode constraint library [12]. A modified version of the BDD library *CUDD* [13] is used internally for the domain representation. The source code of the complete extension is available [9] under the terms of the MIT license.

7.1. Domain approximation

As the domain D_X of a variable X can be a considerably large set of relations we require an approximation of it that can be practically stored. The goal of this approximation is to provide a good trade off between the complexities of representing the data of the domain and the basic operation on this data that supports the operations in the domain. D_X is approximated by the pair of relations $\langle glb, lub \rangle$. The set of represented relations is $\{R : glb \subseteq R \subseteq lub\}$. *glb* stands for *greatest lower bound* and *lub* for *least upper bound*. In the following we use $glb(X)$ (resp. $lub(X)$) when referring to the relation *glb* (resp. *lub*) of D_X. This representation of the domain was proposed in [8] to represent the domain of integer sets.

7.2. Domain and bound consistency

With the formalization of the relation domain as presented in Section 3 it is possible to have constraints that enforce *domain consistency*[12]. Let us consider a constraint C with scope $X = \{X_1, .., X_n\}$. C is domain consistent if for every variable X_i: $\forall r \in D_{X_i}$, $\forall j \neq i, \exists s \in D_{X_j} : C$. That is, for every relation in the domain of every variable there are relations in the domains of the other variables that, considered together, satisfy C.

Achieving domain consistency is not possible for all the constraints under the bound approximation. By enforcing constraints like projection, permutation, inclusion and exclusion on the bounds of the domain it is possible to still achieve domain consistency. As an example of this consider exclusion, after enforcing it on the upper bound all the values in the domain respect the constraint.

Under this representation, the notion of bound consistency is more common: a constraint C with scope $X = \{X_1, .., X_n\}$ enforces *bound consistency*[13]. C is bound consistent if for every variable X_i, $\forall r_i \in \{glb(D_{X_i}), lub(D_{X_i})\}$, $\forall j \neq i, \exists s \in D_{X_j}, C(X_1, X_2, \ldots, glb(D_{X_i}), \ldots, X_n) \wedge$ $C(X_1, X_2, \ldots, lub(D_{X_i}), \ldots, X_n)$.

7.3. Binary decision diagrams

The data structures used to represent the domain of relation variables are *binary decision diagrams* (BDDs), first introduced by [2] and improved later by [6]. BDDs turn out to be a very good data structure for this domain representation. The complexity of basic operations (e.g. union, intersection, quantification, etc.) are defined in terms of the variables of a function and not on the amount of data it represents. Features such as canonicity bring constant complexities to common operations like equality testing. Shared BDDs improve the space complexity of the domain representations of the variables in a CSP. Complemented edges make possible to complement a relation at almost no cost.

There are other variants of decision diagrams such as *algebraic*, *zero suppressed* or even *interval* decision diagrams. Our choice of use just binary decision diagrams is supported by the common use of the complement operation internally and the advantages that we got from the sharing of nodes at the implementation level. However, a

[12] Also called generalized arc consistency.

[13] Also called interval consistency.

careful study and comparison of the other variants has to be done as a future work.

8. CONCLUSION

This paper defines the relation domain in CP to model and solve problems in the musical field. We presented some examples of its application in music composition. For this application we introduced basic notions such as *musical bundle* and *musical bundle sets* as structured ways of representing different musical concepts while keeping a connection with the constraint domain.

The domain is introduced along with general purpose constraints that allow to state properties on decision variables. These constraints are then used to model musical properties on musical entities in a way that is not possible to do directly in finite domains or finite set domains. For instance, we can reason directly with concepts like scores and transformations between scores. The constraint inferencing performed by the CP solver can use this information to improve efficiency.

We do not claim that the problems presented in this paper cannot be tackled by using other domains. However, the models of these problems will involve more representation effort and will likely lead to bigger, less clear and possibly less efficient models. An example of this can be seen in the differences between the two models for the problem of finding a harmonic round presented in Section 1.

An implementation of the domain on top of the Gecode [12] library is available [9]. This allows to start using the domain and constraints in musical applications using C++. An interface to use Gecode (including the relational domain) from OpenMusic [1] is currently work in progress[14].

8.1. Future work

The work we have presented is just the beginning of a wider set of applications of CP on relations in the musical field. We recognize that we can target other areas like orchestration where we are already working. New abstractions in forms of constraints that address particular properties on music are also under consideration. For instance, using musical transformations, we could model *theme and variation* problems. Moreover, using the subset constraint we can consider modeling *leitmotiv* constraints.

The MBSs we used are generally scores but we could consider to work with other types of MBSs, such as set of sounds, where a sound is represented in terms of physical parameters. In the future, we also plan to develop a visual framework inside OpenMusic [1] for composers. This framework should contain built-in constructs such as scores, bars, and other MBSs, some predefined constraints that can be applied on them and some search heuristics. Moreover, it should allow the possibility to be extended with new constraints, constructs and search heuristics. All those concepts should be represented visually, using visual components such as patches.

[14] https://github.com/svancauw/GeLiSo

9. REFERENCES

[1] C. Agon, G. Assayag, and J. Bresson, "OpenMusic," 1998. [Online]. Available: http://repmus.ircam.fr/openmusic/home

[2] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516, 1978. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1675141

[3] T. Anders, "Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System," Ph.D. dissertation, Queen's University, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.9497

[4] T. Anders, C. Anagnostopoulou, and M. Alcorn, "Strasheela : Design and Usage of a Music Composition Environment Based on the Oz Programming Model," *Second International Conference on Multi-paradigm Programming in Mozart/Oz (MOZ 2004)*, vol. 3389, pp. 277–291, 2004. [Online]. Available: http://www.springerlink.com/index/MU6GVAY4Q73PG89K.pdf

[5] T. Anders and E. R. Miranda, "Constraint programming systems for modeling music theories and composition," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–38, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1978802.1978809

[6] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[7] C. J. Date and H. Darwen, *Foundation for object/relational databases: The third manifesto : a detailed study of the impact of objects and type theory on the relational model of data including a comprehensive proposal for type inheritance*. Addison-Wesley (Reading, Mass.), 1998.

[8] C. Gervet, "Conjunto: constraint logic programming with finite set domains," *Proc of ILPS*, no. ECRC-94-15, pp. 339–358, 1994. [Online]. Available: citeseer.nj.nec.com/gervet94conjunto.html

[9] G. Gutiérrez and Y. Jaradin, "CPRelLib: A Relation Domain Constraint Library," 2010. [Online]. Available: http://ggutierrez.github.com/cprelmaster

[10] J. Kretz, "Navigation of Structured Material in Second Horizon for Piano and Orchestra," in *The OM Composer's Book .1*, 2006.

[11] S. Lemouton, "Using Gecode to Solve Musical Constraint Problems," in *Constraint Programming in Music*, C. Truchet and G. Assayag, Eds. Wiley-Blackwell, 2011.

[12] C. Schulte, M. Lagerkvist, and G. Tack, "Gecode: Generic Constraint Development Environment," 2006. [Online]. Available: http://www.gecode.org

[13] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.5.0," 2012. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/