Constraint Programming Meets SQL

Rafael Caballero[*]
University Complutense de Madrid
Spain
rafacr@ucm.es

Carlo Ieva
Simula Research Laboratory
Oslo, Norway
carlo@simula.no

We present a proposal for introducing SQL tuples into the modeling programming language MINIZINC. The domain of the new decision variables is defined by arbitrary relational database tables indicated by the user. The new setting increases the expressiveness of MINIZINC, allowing the modeler to mix the usual finite domains already existing in the language with string constraints typical from SQL such as **`concat`**, **`substr`**, or **`like`**. In order to obtain the solutions of these combined models, we first replace the atomic constraints involving strings by boolean variables. The result is a standard MINIZINC model, which can be solved by any off-the-shelf solver. Then, each individual solution is applied to the remainder string constraints, which are then solved using a SQL query. We discuss how both languages, MINIZINC and SQL, benefit from this combination.

1 Introduction

Constraint programming [8] aims at providing mechanisms that allow the user to represent complex problems in a natural way, solving hard combinatorial problems of practical relevance.

Programs in constraint programming are known as *models*. The constraints in a model describe a problem by restricting the possible values that variables can take. Constraints are typically defined on some specific domain, such as boolean domain,also known as SAT problems[4], integer domains, linear and non-linear domains, or finite domains, where constraints are defined over finite sets.

However, usual constraint modeling systems like MINIZINC[10] do not allow defining constraints over strings. Instead, specific string constraints have been developed. Most of them consider constraints defining membership in regular languages[3] or fixed-size context-free languages [6].

In this paper, we present a different point of view, requiring that the domain of the string variables to range over a SQL table. Our setting aims at introducing SQL tuples as first-class citizens of the constraint modeling language MINIZINC. In the case of tuple attributes that correspond The MINIZINC standard values such as boolean or integers, the standard domain constraints are allowed. In the case of string constraints, the usual SQL constraints such as `concat`, `length`, or **`like`** can be employed.

To the best of our knowledge this is the first work presenting this combination. Other works such as [2, 7] aim at integrating the use of constraints into relational query languages. We think that this new perspective increases the expressiveness and possibilities of constraint modeling languages.

The natural field of application of our setting, denominated MINIZINC+SQLin the rest of the paper, is related to the area of natural language processing [1], specially in the fields of named entity recognition (NER)[9], natural language generation [11], or optical character recognition (OCR) [12].

The structure of the paper is as follows: next section introduces briefly MINIZINC, SQL and the combination proposed in this paper, MINIZINC+SQL. Section 3 presents the solving mechanism that

[*]Work partially supported by the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731) and UCM grant GR3/14-910502.Supported by a grant from Iceland, Liechtenstein and Norway through the EEA Financial Mechanism. Operated by Universidad Complutense de Madrid.

Submitted to:
PROLE 2015

allows obtaining the solutions in the new framework. Section 4 discusses the possibility of using simple SQL instead of MINIZINC+SQLto obtain the same result.The paper ends with some conclusions in Section 5.

2 SQL, MINIZINC and their combination

Before introducing the new modelling language proposed in this paper we introduce briefly both SQL and MINIZINC.

2.1 The query language SQL

The syntax of SQL queries can be found in [5]. A typical query Q contains both **`select`** and **`from`** sections in its definition with the optional **`where`**, **`group by`** and **`having`** sections. In our setting we need only SQL queries with **`select`**, **`from`** and **`where`** sections:

> **select** e_1 $E_1, \ldots, e_n$ E_n
> **from** T_1 N_1 $, \ldots, T_m$ N_m
> **where** C_w;

The **`from`** section indicates that SQL must first compute the Cartesian product of tables $T_1 \ldots T_m$, which are referred in the rest of the query as $N_1, \ldots, N_m$ respectively. The alias N_i are specially useful when the same table is used more than once playing different roles in the same query. In a SQL query the attribute A of a table T with alias N is represented as $N.A$. The **`where`** section defines a condition C_w which filters the rows of the Cartesian product that are displayed finally by the **`select`** section. The values e_i are expressions that combine the values of the selected rows. These values are displayed for each row of Cartesian product verifying C_w. Finally, the names E_i are simply the name displayed by SQL at the top of each column.

The **`where`** condition can contain conjunctions (**`and`**), disjunctions (**`or`**) and negation (**`not`**) of atomic constraints. There also extended possibilities like existential conditions, set operation, or the use of subqueries inside conditions, but they are not necessary here. Regarding the atomic constraints, apart of the usual arithmetic and relational operators, SQL offers an extended variety of primitives such as *concat*, *substr*, *length* or **`like`**. In particular, the primitive **`like`** is very useful for checking if a string has certain regular form. **`Like`** patterns can use '_' to represent any single character and '%' to represent any sequence of characters. For instance the condition `T.A` **`like`**'`%ous`' filters those rows such that the value of attribute A in table T terminates in 'ous'.

2.2 The modeling language MINIZINC

MINIZINC is a constraint modeling language. A typical MINIZINC *model* is depicted in Figure 1. At this point we are only interested in the general structure of the model, the meaning of this particular model will become clear later.

The structure of the MINIZINC models shown in this paper can be divided into three parts:

1. First, some *decision variables* are declared. This example uses arrays of integers (`wtype` and `wsyllable`) and booleans(`b`), as well as individual boolean variables (`a1` and `a2`). The domain of the variables include also sets and floats, but not strings.

```
1  array[1..6] of var int:wtype;
2  array[1..6] of var int:wsyllables;
3  var bool:a1;  var bool:a2;
4  array[1..6] of var bool:b;
5
6  constraint wtype[1]=2 \/ wtype[1]=3;
7  constraint (wtype[1]=2 /\ wtype[2]=1) \/ (wtype[1]=3 /\ wtype[2]=2);
8  constraint wtype[3]=wtype[1] /\  wtype[4]=wtype[2];
9  constraint (wtype[1]=2 /\ wtype[5]=3) \/ (wtype[1]=3 /\ wtype[5]=2);
10 constraint (wtype[5]=2 /\ wtype[6]=1) \/ (wtype[5]=3 /\ wtype[6]=2);
11 constraint forall(i in 1..6)(wsyllables[i]>0);
12 constraint forall(i in 1..3)(wsyllables[2*i-1] + wsyllables[2*i]=6);
13 constraint a1 /\ a2;
14 constraint forall(i in 1..3)
15            ((wtype[2*i]=2 /\ b[2*i-1]) \/
16             (wtype[2*i]=1 /\ b[2*i]));
17 solve satisfy;
```

Figure 1: MINIZINC model

2. The **`constraint`** statements restrict the domain of the variables. For instance, line 6 indicates that the integer decision variable `wtype[1]` can only take values 2 or 3.
3. Finally, the **`solve`** statement indicates that this is a satisfaction problem, meaning that we look for solutions that satisfy the model[1]. These solutions are represented in the rest of the paper by θ or σ and are substitutions of decision variables by values in their domain that verify all the constraints of the model. If θ is a solution and x a decision variable, we denote by $\theta(x)$ the value taken by x in the solution θ. In the case of an expression e we use also the notation $\theta(e)$ to indicate the evaluation of e obtained after replacing every variable $x \in e$ by $\theta(x)$.

Regarding constraints, we assume that a MINIZINC constraint c can be easily translated into an equivalent SQL **`where`** condition, represented by $\parallel c \parallel$. Although we have not proved formally the existence of this transformation it seem to hold in all the cases we have checked.

2.3 The combined setting MINIZINC-SQL

Figure 2 includes an example of a MINIZINC+SQL model that aims at obtaining stanzas of three lines or verses, each one with two words:

word1 word2,
word3 word4,
word5 word6

In particular, we look for two types of stanzas according to the type of the words:

adjective	noun	adverb	adjective
adjective	noun	adverb	adjective
adverb	adjective	adjective	noun

[1]MINIZINC also allows maximization problems, and our proposal is also valid in this case.

```
table  words = 'words'

array[1..6] of var words:w;

% types of words
constraint (w[1].type=2 \/ w[1].type=3);
constraint (w[1].type=2 /\ w[2].type=1) \/
            (w[1].type=3 /\ w[2].type=2);
constraint w[3].type=w[1].type /\  w[4].type=w[2].type;
constraint (w[1].type=2 /\ w[5].type=3) \/
            (w[1].type=3 /\ w[5].type=2);
constraint (w[5].type=2 /\ w[6].type=1) \/
            (w[5].type=3 /\ w[6].type=2);

% metre
constraint forall(i in 1..6)(w[i].syllables>0);
constraint forall(i in 1..3)(w[2*i-1].syllables + w[2*i].syllables = 6);

% different words
constraint alldifferent([w[1].word,w[3].word,w[5].word]);
constraint alldifferent([w[2].word,w[4].word,w[6].word]);

% rhyme: adjectives must end in 'ous', adverbs in 'i_e'
constraint forall(i in 1..3)
            (w[2*i].type=2 /\ w[2*i].word like '%ous')
             \/
            (w[2*i].type=1 /\ w[2*i].word like '%i_e');

solve satisfy;
```

Figure 2: MINIZINC model including SQL tuples

Moreover, the words must be obtained from a SQL table *words*. This is represented in the model by the first line that declares the table.[2] Each record in this table has three attributes:

- *word*: A string. This is the primary key of the table.
- *type*: an integer representing the type of word. This example uses three types:
  - Type=1 means that the word is a noun.
  - Type=2: adjective.
  - Type=3: adverb.
- *syllables*: number of syllables of the word.

[2]Required information necessary to connect to the database such as URL, user, password, etc. is assumed to be provided during the compilation of the model.

Line 3 of Figure 2 declares the 6 tuples that constitute the stanza as an array. Lines 6-13 define the two type of stanzas allowed depending on the type of the words. Line 16-17 constrain the verses to have six syllables each one. Line 20 indicate that the first words of each verse are pairwise different using the constraint `alldifferent`. This predicate exists as a global constraint in MINIZINC, and we extend it to strings in our new setting. Analogously, line 21 do the same with the second words of the verses. Finally, lines 24-27 indicate that the nouns finishing a verse must end in 'i_e' with _ any letter, while the adjectives must end in 'ous'. The predicate **`like`** employed in this constraint is not part of standard MINIZINC, and establishes an atomic string constraint.

3 Solving the new models

Let M^S be a MINIZINC+SQL model. Our goal in this section is to devise a technique for obtaining all the solutions of M^S.

3.1 First phase: MINIZINC

word index	Type	Syllables	New boolean variable b
1	3 (adverb)	3	true (w[2].word like '%ous')
2	2 (adjective)	3	false (w[2].word not like 'i_e')
3	3 (adverb)	4	true (w[4].word like '%ous')
4	2 (adjective)	2	false (w[4].word not like 'i_e')
5	2 (adjective)	3	false (w[6].word not like '%ous')
6	1 (noun)	3	true (w[6].word like 'i_e')

Figure 3: Solution to standard MINIZINC model of Figure 1

The first step is to replace all the atomic constraints in M^S that do not correspond to MINIZINC standard models by new boolean decision variables. This can be done automatically by the compiler. After this replacement we obtain a standard MINIZINC model M.

The transformation of the example in Figure 2 is shown in Figure 1. In this MINIZINC model the boolean variables `a1` and `a2` represent the two `alldifferent` constraints in the M^S model. For i between 1 and 3, $b[2i-1]$ indicates if the word $2i$ ends in 'ous', while $b[2i]$ indicates if the same word ends in 'i_e'. Observe that from every solution θ_{M^S} of M^S we can obtain a solution σ_M of M in the following way:

1. The variables x of M that are also in M^s (that is the standard variables in M^S) take the same value in both solutions, that is $\sigma_M(x) = \theta_{M^S}(x)$.
2. The value in S of each new boolean variable $y \in M$ obtained after the replacement of some string atomic constraint $c \in M^s$, is defined as $\sigma_M(y) = \theta_{M^S}(c)$, that is, as the result `true` or `false` obtained after solving the constraint under the variable assignments included in θ_{M^S}.

In this sense we can say that

$$Solutions(M^S) \subset Solutions(M)$$

However, the converse is not true

$$Solutions(M) \not\subseteq Solutions(M^s)$$

The reason is that the solution for MiniZinc might not correspond to any real values in the SQL tables. For instance, Table 3 displays one of the 2000 solutions found by the MiniZinc solver for the M model of Figure 1. Notice that we do not include the boolean variables `a1` and `a2` which are assigned to *true* in every solution of the model. In the case of the type and of the new variables $b[i]$ the table includes an explanation between round brackets. This solutions indicates that a possibility is to form the first verse (indexes 1 and 2) starting with an adverb of three syllables followed by an adjective also of three syllables and with termination 'ous'. However, we cannot ensure that the SQL table *words* contains words of these characteristics. This is checked in the second phase of the solving process.

3.2 Second phase: SQL

In this phase each solution σ obtained from the standard MiniZinc model M is checked and completed using a SQL query. The query is constructed as follows:

- The **`from`** section includes one table for each tuple in the model M^S. The alias of each table is name of the tuple decision variable to avoid name clashing.
- The **`select`** section includes the primary keys of the tuples corresponding to each table introduced in the **`from`** section. This allows identifying the values of the tuple that complete the solution.
- The **`where`** section is of the form `W1 AND W2`, with:
  - `W1` a representation of σ restricted to the variables corresponding to the SQL attributes of standard MiniZinc types in M^S.
  - `W2` is the obtained as follows. First, consider all the constraints $c_1,...,c_n$ in M^S that include atomic string constraints. Then, the atomic string constraints are replaced by their negation if the associated boolean variable x in M verifies $\sigma(x)$ =*false*. Let c'_1, ..., c'_n be the new constraints obtained after this process. Finally, `W2` is defined as $\parallel \sigma c'_1 \parallel$ **`and`** $\ldots$ **`and`** $\parallel \sigma c'_n \parallel$.

It is worth noticing that in the definition of `W2` the application of the substitution $\sigma c'_i$ only affects to the standard variables common to M and M^S. The rest of the domain of σ, which consists of the new boolean variables introduced in the first phase, and cannot occur in c'_i. Thus, after this substitution the only 'free' attribute names occurring in `W2` (and thus in the **`where`** section) correspond to the string attributes. The goal of this query is precisely to find suitable values for these attributes in the SQL table if they exist.

The SQL query corresponding to the solution in Table 3 is displayed in Figure 4. Lines 7-11 correspond to the definition of `W1`, that is the assignment of the integer variables contained in σ, while lines 12-20 correspond to the definition of `W2`. In particular, lines 12-14 correspond to the representation of the two `alldifferent` constraints, while lines 15-20 express the constraints including the **`like`** string constraint. This part of the condition can be easily simplified: for instance `(w2.type=1 and w2.word not` **`like`** `'%i_e')` is simply *false* since the part `W1` establishes that `w2.type=2`. However we have kept the redundant conditions in order to explain better the generation of the query.

Running this query produces among many other the following stanza:

Healthily barbarous,
Delightedly aurous,
Balsamic dentifrice!

```
select w1.word w1, w2.word w2, w3.word w3, w4.word w4,
       w5.word w5, w6.word w6
from words w1, words w2, words w3, words w4,  words w5, words w6
where
     w1.type=3 and  w2.type=2 and  w3.type=3 and
     w4.type=2 and  w5.type=2 and  w6.type=1 and
     (w1.syllables=3 and  w2.syllables = 3) and
     (w3.syllables=4 and  w4.syllables = 2) and
     (w5.syllables=3 and  w6.syllables = 3) and
     w1.word<>w3.word and w1.word<>w5.word and
     w3.word<>w5.word and w2.word<>w4.word and
     w2.word<>w6.word and w4.word<>w6.word  and
     ((w2.type=2 and w2.word like '%ous') or
      (w2.type=1 and w2.word not like '%i_e')) and
     ((w4.type=2 and w4.word like '%ous') or
      (w4.type=1 and w4.word not like '%i_e'))and
     ((w6.type=2 and w6.word not like '%ous') or
      (w6.type=1 and w6.word like '%i_e'))
```

Figure 4: SQL query corresponding to the solution displayed in Table 3

In order to improve the (always subjective) quality of the poems there are many factors that can be taken into account, like distinguishing between unstressed syllables and stressed syllable, etc. However, such improvements are beyond the scope of this paper where it is used as a simple running example.

4 SQL performance

At this point the improvement in the expressiveness of MINIZINC+SQL with respect to the standard MINIZINC should be clear. Models similar to that of Figure 2 are not possible to represent in standard MINIZINC. However, a SQL programmer might wonder which are the benefits of using MINIZINC+SQL instead of simply representing the problem in SQL. In the case of our running example, we could simply write the query displayed at Figure 5.

The query uses the Oracle function **decode** to abbreviate a combination of conjunctions and disjunctions. The problem with this query is performance. After 4 hours no answer is obtained in a laptop computer[3].

However, in the case of the MINIZINC+SQL the first phase (generating the standard MINIZINC model and solving it) required less than 2 seconds in the same computer, while the first SQL query starts displaying results in less than one second.

The reason behind this big difference in terms of performance is that conditions like `(w1.syllables + w2.syllables = 6)` prevent the use of indices even in a well-tuned database. Instead, the query of Figure 4 includes atomic conditions of the form `(w1.syllables=3` **`and`** `w2.syllables = 3)` that allows using the indices. Thus, using first MINIZINC benefits from the technique used in constraint

[3]Database Oracle 12.1 running on a Processor 4xIntel(R) Core(TM) i7-2640M CPU @ 2.80GHz with 8Gb of memory and Operating System Ubuntu 14.04

programming for obtaining readily the possible values of integers, while including them directly in the SQL requires the full scan of the tables looking for the solution of the finite domains.

```
select w1.word w1, w2.word w2, w3.word w3, w4.word w4,
       w5.word w5, w6.word w6
from words w1, words w2,
     words w3, words w4,
     words w5, words w6
where (w1.type = 2 or w1.type=3) and
     w2.type=decode(w1.type,2,1,3,2) and
     w3.type=w1.type and
     w4.type=w2.type and
     w5.type=decode(w1.type,2,3,3,2) and
     w6.type=decode(w5.type,2,1,3,2) and
     (w1.syllables + w2.syllables = 6) and
     (w3.syllables + w4.syllables = 6) and
     (w5.syllables + w6.syllables = 6) and
     w1.word<>w3.word and w1.word<>w5.word and
     w3.word<>w5.word and
     w2.word<>w4.word and w2.word<>w6.word and
     w4.word<>w6.word and
     ((w2.type=2 and w2.word like '%ous') or
      (w2.type=1 and w2.word like '%i_e'))     and
     ((w4.type=2 and w4.word like '%ous') or
      (w4.type=1 and w4.word like '%i_e'))     and
     ((w6.type=2 and w6.word like '%ous') or
      (w6.type=1 and w6.word like '%i_e'));
```

Figure 5: SQL query solving the problem represented by model Figure 2

5 Conclusions

Strings are ubiquitous in computer science and integrating them into a general-purpose constraint modeling language such as MiniZinc entails an improvement in the language expressiveness. Moreover, we have shown that the combination is also positive with respect to the direct use of a query language in terms of query performance.

We have also presented an effective way of solving the constraints in the new setting. Although we have chosen SQL and MiniZinc as database and modeling language respectively, the same schema presented here is applicable to other databases (for instance NoSql databases) and constraint languages (for instance Gecode).

The more severe limitation of the approach is the great number of SQL queries that can be required. In our running example 2000 queries were needed. All of them are solved readily by the database system assuming a table with indices on the integer and string fields, because our technique produces queries with atomic conditions of the form *attribute=constant*. However, many of the queries are equivalent and could be avoided. A future line of search is to detect and eliminate equivalent queries.

References

[1] M Bates (1995): *Models of natural language understanding*. Proceedings of the National Academy of Sciences 92(22), pp. 9977–9982. Available at `http://www.pnas.org/content/92/22/9977.abstract`.

[2] Marco Cadoli & Toni Mancini (2007): *Combining Relational Algebra, SQL, Constraint Modelling, and Local Search*. Theory and Practice of Logic Programming 7(1-2), pp. 37–65, doi:10.1017/S1471068406002857.

[3] Keith Golden & Wanlin Pang (2003): *Constraint reasoning over strings*. In: Principles and Practice of Constraint Programming–CP 2003, Springer, pp. 377–391.

[4] Frank van Harmelen, Frank van Harmelen, Vladimir Lifschitz & Bruce Porter (2007): *Handbook of Knowledge Representation*. Elsevier Science, San Diego, USA.

[5] ISO/IEC (2011): *SQL:2011 ISO/IEC 9075(1-4,9-11,13,14):2011 Standard*.

[6] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer & Michael D. Ernst (2009): *HAMPI: A Solver for String Constraints*. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, ACM, New York, NY, USA, pp. 105–116, doi:10.1145/1572272.1572286.

[7] Toni Mancini, Pierre Flener & Justin K. Pearson (2012): *Combinatorial Problem Solving over Relational Databases: View Synthesis Through Constraint-based Local Search*. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, ACM, New York, NY, USA, pp. 80–87, doi:10.1145/2245276.2245295.

[8] K. Marriott & P.J. Stuckey (1998): *Programming with Constraints: An Introduction*. Adaptive Computation and Machine, MIT Press. Available at `https://books.google.es/books?id=jBYAleHTldsC`.

[9] David Nadeau & Satoshi Sekine (2007): *A survey of named entity recognition and classification*. Linguisticae Investigationes 30(1), pp. 3–26. Available at `http://www.ingentaconnect.com/content/jbp/li/2007/00000030/00000001/art00002`. Publisher: John Benjamins Publishing Company.

[10] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck & Guido Tack (2007): *MiniZinc: Towards a standard CP modelling language*. In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming, Springer, pp. 529–543.

[11] Ehud Reiter & Robert Dale (2000): *Building Natural Language Generation Systems*. Cambridge University Press, New York, NY, USA.

[12] S.V. Rice, G. Nagy & T.A. Nartker (1999): *Optical Character Recognition: An Illustrated Guide to the Frontier*. The Springer International Series in Engineering and Computer Science, Springer US, doi:10.1007/978-1-4615-5021-1.