Tabular Constraint-Satisfaction Problems and Answer-Set Programming

Raphael A. Finkel, Victor Marek and **Mirosław Truszczyński**
Department of Computer Science
University of Kentucky
Lexington KY 40506-0046, USA
email: raphael|marek|mirek@cs.uky.edu

Abstract

We argue that database query languages can be used to specify constraint satisfaction problems. We describe one such language here. It is based on tuple relational calculus and its *sentences* are used to define constraints. For a fragment of this language we develop a front end, Constraint Lingo, to facilitate programming tasks. Finally, we show that programs in Constraint Lingo can be compiled into DATALOG$^\neg$ and its versions and processed by ASP solvers such as *smodels*.

Introduction

In this preliminary report, we treat the class of constraint-satisfaction problems that can be conveniently modeled by means of query languages developed in the area of databases. We discuss one such language here based on tuple relational calculus (Maier 1983). We describe an implementation of a restricted version that we call Constraint Lingo. We show that answer-set programming formalisms such as *smodels* (Niemelä & Simons 1997; 2000), *dcs* (East & Truszczyński 2000) and *dlv* (Eiter *et al.* 1998) can be used as a processing back end for Constraint Lingo. We conclude with a brief discussion of related research problems.

DATALOG$^\neg$ is a restricted version of the formalism of logic programming with stable-model semantics in that it disallows function symbols from the language. It has been proposed as a query language; the restriction is motivated by the requirement of finiteness of answers to database queries (Ullman 1988). A database, say D, is represented in DATALOG$^\neg$ by a collection of ground atoms defining extensions of the corresponding predicates. These atomic facts are referred to as the *extensional* database and their predicate symbols as *extensional* predicates. A DATALOG$^\neg$ program Q such that extensional predicates appear only in the bodies of its rules is called a *query*. All non-extensional predicates occurring in a query are called *intensional*. A query Q on an extensional database D is a specification of intensional predicates appearing in Q. The extension of one (distinguished) intensional predicate serves as the answer to Q. This extension is defined through some chosen semantics of logic programs with negation, for instance, stable-model semantics. Namely, if M is a stable model of $Q \cup D$, and $answer$ is the intensional predicate of interest, the answer is given by

$$\{(a_1, \ldots, a_k) \colon answer(a_1, \ldots, a_k) \in M\}.$$

While DATALOG$^\neg$ has received some attention in the database community, it has never become a mainstream query language. General DATALOG$^\neg$ programs are not guaranteed to have a unique stable model, so queries are not guaranteed to fully specify the answer, a highly undesirable feature from the database perspective.

We argue in (Marek & Truszczyński 1999) that DATALOG$^\neg$ is more appropriately viewed as a formalism to specify constraint-satisfaction problems than as a database-query language. (A similar argument has been made by Niemelä (Niemelä 1999).) That is, a query Q specifies constraints on (unknown) intensional predicates. These constraints also involve some known predicates defined by the extensional database D. Stable models of $Q \cup D$ define solutions — possible instantiations of unknown predicates satisfying the constraints. For instance, the problem of existence of a Hamilton cycle can be cast in these terms (Marek & Truszczyński 1999). Two extensional (and fully defined) predicates `vtx` and `edge` describe the graph. An intensional (unknown) predicate `hc` describes the edges of the Hamilton cycle. All the constraints on `hc` (and some other auxiliary intensional predicates) are given in a DATALOG$^\neg$ program (query). Multiple stable models do not pose a problem in such use of DATALOG$^\neg$ — they simply determine different ways to satisfy constraints. We refer to DATALOG$^\neg$ and other logics with similar features as *answer-set programming* (ASP) formalisms.

Solutions to constraint-satisfaction problems defined by DATALOG$^\neg$ programs are collections of tuples. It is often convenient to think about these solutions as tables. Consequently, we refer to them as *tabular*. In this paper we study tabular constraint-satisfaction problems (tCSPs). We are interested both in modeling tabular CSPs (programming) and in processing (solving) them. DATALOG$^\neg$ lacks explicit means to model high-level constraints that rely on a tabular structure of solutions and that often can be conveniently phrased in standard database terminology. Examples of such constraints include equalities and inequalities of sets of tuples defined by logical expressions (or database queries). In this paper, we study languages to allow us specify such constraints. In particular, we show that standard database query languages can be used to this end, *although queries must be interpreted differently from when they are used for querying databases.*

We focus our discussion on one database query language, based on tuple relational calculus (Maier 1983). We show that its sentences can be viewed as constraints on predicates (tables). Consequently, tuple relational calculus can serve as a language to define tabular CSPs. One can show that tuple relational calculus can be regarded as a fragment of DATALOG$^{\neg}$ (the details will be included in the full version of this paper). That is, every specification of a tabular CSP given in tuple relational calculus can be translated, in polynomial time in its size, into DATALOG$^{\neg}$.

For a fragment of tuple relational calculus we define a high-level language, called Constraint Lingo, that simplifies the process of defining constrains. Constraint Lingo is designed to provide the user with constructs explicitly supporting commonly occurring constraints and using a simple and straightforward syntax. Thus, Constraint Lingo can be viewed as a high-level front end for a fragment of a constraint-definition language based on tuple relational calculus.

Constraint Lingo is not only a specification language for constraints; its programs can be processed. We show how Constraint Lingo (more generally, tuple relational calculus) can be translated into DATALOG$^{\neg}$. We have implemented this translation; DATALOG$^{\neg}$ solvers such as *smodels* (Niemelä & Simons 1997; 2000) can be used as back-end engines for processing Constraint Lingo programs and solving (some) tabular CSPs. We briefly comment on our computational experience with Constraint Lingo.

This is a preliminary report. Our next goal is to extend Constraint Lingo so that it can support modeling all constraints that can be stated in terms of tuple relational formulas.

Tuple relational calculus and the formalization of tCSP

We first show how some tabular CSPs can be specified using a version of the familiar language of tuple relational calculus (Maier 1983). In the terminology of mathematical logic, this language is *typed*, that is, all objects have *types*. In view of the applications we have in mind (specification of tCSPs), we focus on a language where there is one type called *tuple*. Other types are used to describe *attributes*. The set of attributes is denoted by $\mathcal{A}$.

Each attribute $\mathtt{a}$ in the set of attributes $\mathcal{A}$ is a type with domain $D_{\mathtt{a}}$. We assume that $D_{\mathtt{a}}$ is finite. We assume domain-closure axioms (DCA) for each type. The formalization of those axioms is given below.

We assume that for each attribute $\mathtt{a}$ in $\mathcal{A}$ we have an operation $x.\mathtt{a}$ defined for terms of type *tuple* with values in the type $\mathtt{a}$.

Now we define terms. Each term has a type. Terms of type *tuple* are tuple variables. Terms of type $\mathtt{a}$ (where $\mathtt{a}$ is an attribute) are expressions of the form $x.\mathtt{a}$ where x is a variable of type *tuple* and constants b from $D_{\mathtt{a}}$.

We have *built-in relation symbols* (Maier calls them *comparators*), including $=$ and $\neq$[1], with other built-in relation symbols allowed so long as values of those are tabled or predefined. For instance, numeric attributes have $<$ and $\leq$ as built-in relation symbols. In the example below we see other built-in symbols.

There are no variables for types other than *tuple*; variables for any other type are *obtained* by applying operations $x.\mathtt{a}$ to variables of type *tuple*.

Let us fix the attributes, their domains, and built-in relations. We define a language $\mathcal{L}$ as follows. Atoms of the language $\mathcal{L}$ are expressions of the form: $c(t_1, \ldots, t_k)$ where c is a built-in relation symbol of our language and $t_1, \ldots, t_k$ are terms of appropriate types (every place in the symbol c has an associated type). Atoms can also be of the form $answer(s)$, where *answer* is a unique unary predicate symbol. The idea is that the predicate *answer* is *specified* by the constraints of our tCSP. Formulas are defined from atomic formulas by the usual induction, with Boolean operators and quantifiers.

Now we define the semantics of our language. The models are pairs $\langle U, T\rangle$, where U is the set of all tuples s with the property that the values of s on attribute $\mathtt{a}$ belong to $D_{\mathtt{a}}$, and T is a subset of U, that is a unary relation in U. This table T interprets the predicate $answer$.

We now define the satisfaction relation $\models$ between the triples of the form $\langle\langle U, T\rangle, \varphi, s\rangle$, where s is a function that assigns to all the variables occurring in φ tuples from the set U. We write the relation as $\langle U, T\rangle \models_s \varphi$.

It suffices to define satisfaction for atomic formulas; satisfaction for other formulas can be defined in the usual, Tarskian, manner. To this end, we first define the value of terms occurring in the formula φ under valuation s. The value of term t under valuation s is denoted t_s. When v is a variable (and thus denotes a tuple), then v_s is defined as $v(s)$ (which is well-defined, as v occurs in φ). Then, for all other types we define:

$$v.\mathtt{a}_s = v(s)(\mathtt{a})$$

and $a_s = a$ when a is a value of the attribute $\mathtt{a}$.

We proceed to define the satisfaction relation $\models_s$ where s is a valuation. First, we give the definition of satisfaction for atomic formulas.

1. $\langle U, T\rangle \models_s answer(x)$ if $s(x) \in T$
2. $\langle U, T\rangle \models_s t_1 = t_2$ if $(t_1)_s = (t_2)_s$ (and analogously for $\neq$)
3. $\langle U, T\rangle \models_s r(t_1, \ldots, t_k)$ if $\langle (t_1)_s, \ldots, (t_k)_s\rangle \in R$ where R is the relation interpreting the built-in relational symbol r

The definition of $\models_s$ uniquely extends to the satisfaction relation for all formulas of $\mathcal{L}$.

Satisfaction of formula φ in table T by valuation s depends *only* on values of s on variables that are free in φ. Thus for *sentences* φ, that is, formulas without free variables, either all or no valuations satisfy φ in T. We write $\langle U, T\rangle \models \varphi$ when valuations v satisfy φ in $\langle U, T\rangle$. Similarly, $\langle U, T\rangle \models C$ where C is a set of constraints, when for all $\varphi \in C$, $\langle U, T\rangle \models \varphi$.

We intend that the sentences of $\mathcal{L}$ serve as *constraints* on the table that is the solution to the tCSP. Thus, when our

[1] Technically, we should have separate relations of equality and inequality for each type.

constraints are described by a set of sentences, say, C, then our goal is to find the table(s) T such that $\langle U, T\rangle \models C$. Since the table T describes the predicate $answer$, this table has precisely the properties required by the set of constraints C.

We show now how this language allows for formalization of desired constraints. First, we formalize the domain-closure axiom DCA that we always assume.

$$DCA_{\mathtt{a}}: \qquad \forall x \bigvee \{x.\mathtt{a} = \mathtt{b} : \mathtt{b} \in \mathtt{D_a}\}$$

Since $D_{\mathtt{a}}$ is finite, this formula belongs to $\mathcal{L}$. Now we define:

$$DCA = \bigwedge \{DCA_{\mathtt{a}} : \mathtt{a} \in \mathcal{A}\}$$

Second, we formalize the constraint AD, *all_different*. To this end, we define the sentence

$$AD_{\mathtt{a}} : \forall x \forall y (answer(x) \wedge answer(y) \wedge x.\mathtt{a} = \mathtt{y.a} \Rightarrow \mathtt{x} = \mathtt{y})$$

This constraint requires that all tuples in T have different values on attribute a. In other words, a is a candidate key for the table T. Next, we define

$$AD = \bigwedge \{AD_{\mathtt{a}} : \mathtt{a} \in \mathcal{A}\}$$

Third, we formalize the constraint AU, *all_used*.

$$AU_{\mathtt{a}}: \qquad \bigwedge \{\exists x (answer(x) \wedge x.\mathtt{a} = \mathtt{b}) : \mathtt{b} \in \mathtt{D_a}\}$$

Finally, we define

$$AU = \bigwedge \{AU_{\mathtt{a}} : \mathtt{a} \in \mathcal{A}\}$$

We intend to study the tables in the class ADU, that is, those satisfying both AD and AU.

The language $\mathcal{L}$ can be used either as a query language for relational databases or as a constraint language. In the first case, the table T is fixed, and the formula φ contains variables. In this case, our goal is to find $\{t : \langle U, T\rangle \models \varphi[t]\}$, that is, the answer to the query. By contrast, when we use $\mathcal{L}$ as a constraint language, the formulas are sentences, and a collection C of constraints determines $\{T : \langle U, T\rangle \models C\}$. This latter set consists of tables, not tuples. Under the DCA assumption, all these tables are subsets of $D_{\mathtt{a}_1} \times \ldots \times D_{\mathtt{a}_k}$. Each table satisfying the set of sentences C is an equally good solution to the problem formalized by C [2].

We now show how the language $\mathcal{L}$ can be used to formalize a logical puzzle. Specifically, we formalize the puzzle called "French Phrases, Italian Soda"[3]. The following is the wording of the puzzle.

Claude looks forward to every Wednesday night, for this is the night he can speak in his native language to the other members of the informal French club. Last week, Claude and five other people (three women named Jeanne, Kate, and Liana, and two men named Martin and Robert) shared a circular table at their regular meeting place, the Café du Monde. Claude found this past meeting to be particularly interesting, as each of the six people described an upcoming trip that he or she is planning to take to a different French-speaking part of the world. During the discussion, each person sipped a different flavor of Italian soda, a specialty at the café. Using the illustration and the following clues, can you match each person with his or her seat (numbered one through six [circularly]) and determine the flavor of soda that each drank, as well as the place that each plans to visit?

1. The person who is planning a trip to Quebec, who drank either blueberry or lemon soda, didn't sit in seat number one.
2. Robert, who didn't sit next to Kate, sat directly across from the person who drank peach soda.
3. The three men are the person who is going to Haiti, the one in seat number three, and Claude's brother.

To illustrate the formalization process, we have four attributes, which we list with their respective domains.

- *person: claude, jeanne, kate, liana, martin, robert*
- *position: 1, 2, 3, 4, 5, 6*
- *soda: blueberry, grapefruit, kiwi, lemon, peach, tangelo*
- *visits: belgium. haiti, ivory, martinique, quebec, tahiti*

The conditions of the problem suggest that we introduce the following abbreviations:

- $man(x) \Leftrightarrow x.person = claude \vee x.person = martin \vee x.person = robert$
- $even(x) \Leftrightarrow x.position = 2 \vee x.position = 4 \vee x.position = 6$.

Next, the attribute *position* uses these two built-in relations.

- *nexto* with the diagram: $nexto(1, 2), nexto(2, 1), nexto(2, 3), nexto(3, 2), \ldots$ $nexto(6, 5), nexto(6, 1), nexto(1, 6)$,
- *opposite* with the diagram: $opposite(1, 4), opposite(4, 1), opposite(2, 5)$, $opposite(5, 2), opposite(3, 6), opposite(6, 3)$.

The assumption of ADU may simplify some constraints we write. We now write the formulas encoding the constraints (1–3).

1. $\exists x(answer(x) \wedge x.visits = quebec \wedge (x.soda = blueberry \vee x.soda = lemon) \wedge x.position \neq 1)$
2. $\forall x, y(answer(x) \wedge answer(y) \Rightarrow (x.person = robert \wedge y.person = kate \Rightarrow \neg nexto(x.position, y, position)))$.
   $\forall x, y(answer(x) \wedge answer(y) \Rightarrow (x.person = robert \wedge y.soda = peach \Rightarrow opposite(x.position, y.position)))$.
3. $\forall x(answer(x) \wedge x.visits = tahiti \Rightarrow man(x))$.
   $\forall x(answer(x) \wedge x.position = 3 \Rightarrow man(x))$.
   $\forall x, y(answer(x) \wedge answer(y) \wedge x.visits = tahiti \wedge y.position = 3 \Rightarrow x \neq y)$.

[2]To see this fact in the perspective of mathematical logic, the proper analogy is between the use of the language of predicate calculus as a means to specify definable subsets of the domain and the use of theories in the same language to specify elementary classes of models.

[3]Copyright 1999, Dell Magazines; quoted by permission. The original puzzle has nine clues, of which we formalize only three. The remaining six clues can be formalized similarly. By omitting some clues we increase the number of solutions. The original puzzle with all nine clues has a single solution.

$$\forall x, y, z(answer(x) \wedge answer(y) \wedge answer(z) \wedge man(x) \wedge man(y) \wedge man(z) \wedge x \neq y \wedge x \neq z \wedge y \neq z \wedge x.visits = tahiti \wedge y.position = 3 \Rightarrow z.person \neq claude).$$

The theory consisting of DCA, ADU, and the sentences formalizing clauses (1–3) describes precisely those tables that are solutions to the puzzle. We show one solution to this puzzle when we formalize it in the Constraint Lingo programming language.

Constraint Lingo

In the remainder of this paper we describe Constraint Lingo, an implementation of a fragment of the language presented in the previous section. More precisely, we describe a high-level *front end* to model tabular CSPs that can be specified by a fragment of the language of tuple relational calculus determined by a restricted class of queries.

We designed the present prototype of Constraint Lingo bottom up, experimenting with problems and puzzles of increasing difficulty and expanding its functionality. The description we give here is rather informal and is illustrated by a discussion of an example code given in Figure 1. A precise description of Constraint Lingo and its extensions will be given elsewhere.

```
 1  CLASS person: claude jeanne kate liana martin robert
 2  CLASS position: 1 .. 6 circular
 3  CLASS soda: blueberry lemon peach tangelo
 4              kiwi grapefruit
 5  CLASS visits: quebec tahiti haiti martinique
 6                belgium ivory
 7  PARTITION gender: man woman
 8
 9  AGREE man: claude martin robert           # puzzle
10  AGREE woman: jeanne kate liana            # statements
11
12  CONFLICT quebec 1                         # clue 1
13  REQUIRED quebec blueberry OR quebec lemon
14
15  OFFSET !+-1 position: robert kate         # clue 2
16  OFFSET +-3 position: robert peach
17
18  VAR x                                     # clue 3
19  AGREE man: haiti 3 x
20  CONFLICT haiti 3 x
21  CONFLICT x claude
```

Figure 1: Constraint Lingo program for the relaxed version of "French Phrases, Italian Soda" puzzle

The syntax of Constraint Lingo is line-oriented. The character # introduces a comment that continues to the end of the line. For convenience, we call every line of Constraint Lingo a **statement**.

Constraint Lingo allows the user to specify two types of attributes referred to as `CLASS` and `PARTITION`. Attributes marked `CLASS` satisfy constraints all_diff and all_used.

In Figure 1, lines 1, 2, 3, 5, and 7 introduce the relevant *attributes*: `person`, `position`, `soda`, `visits`, and `gender`. The `position` class in line 2 is numeric; Constraint Lingo allows a numeric specification based on its two extreme values. The `circular` specifier indicates that this numeric range is meant to be taken with modular arithmetic; the next "higher" value after 6 is 1.

Domain values must be unique to their domains. Attributes whose order relationships are significant must be represented by numbers. If this puzzle also required days of the week, we could not use 1..7 to represent these values, but we might choose 11..17.

The keyword `PARTITION` in line 7 indicates that the attribute `gender` does not satisfy the all_diff and all_used constraints: multiple tuples may agree on particular values, such as `man`, and it could happen that a particular value, such as `woman`, appears in no row in the solution table at all.

The main programming task is to represent in Constraint Lingo the constraints of the problem. The primary convention of Constraint Lingo is that rows in tables are referred to by elements of domains. Since all domains are disjoint, if we limit ourselves to domains satisfying the all_diff and all_used constraints, for every attribute a and for every element $v \in D(a)$, there is exactly one row in every solution table that has value v in column a.

Domain elements and variables that stand for domain elements (or, equivalently, for table rows) are used to form lists. A list is formed by enumerating its elements.

To stipulate that all domain values that form a list `L` describe the same table row, we list its elements as a line in the program (starting it with the keyword `REQUIRED`). Thus, writing

```
REQUIRED claude quebec
```

means that `claude` and `quebec` must be in the same row (or, in plain English, that Claude visits Quebec).

To indicate that no two elements of a list `L` specify the same row, we write

```
CONFLICT L
```

Line 12 of the code in Fig. 1 indicates that the person going to Quebec is not in position 1. Similarly, line 20 ensures that `haiti`, 3 and `x` represent three different rows in the table.

Given a value `V` in a partition (that is, a domain not satisfying all_diff) and a list `L`, we write

```
AGREE V: L2
```

to stipulate that the row represented by every element of list `L` must agree on value `V`. Line 9 stipulates that the tuples representing three of the people must have `man` in their `gender` attribute; line 10 stipulates that the other three tuples must have `woman`.

We can enforce that two lists refer to the same rows. If `L1` and `L2` are lists,

```
MATCH L1: L2
```

requires that each element on list `L1` be "matched" with an element on list `L2`, and conversely. (It also implies `CONFLICT L1` and `CONFLICT L2`.)

Atomic expressions involving lists can be combined by means of the `OR` connective, indicating that at least one of these conditions holds. Thus, line 13 defines a constraint that the person going to Quebec drinks either blueberry or lemon soda.

Constraint Lingo supports the use of variables to refer to rows in tables. They are interpreted as existentially quantified. To declare a variable with the name var_name, we write:

```
VAR var_name
```

Once declared, the variable can be used in constraints. If a variable x appears in constraints $C_1, \ldots, C_n$ of a program, the interpretation is that there exists a row (referred to by x) that satisfies all constraints $C_1, \ldots, C_n$.

For instance, the variable x defined in line 18 stands for a table row. The interpretation of lines 18–21 is that there is a row in the table that satisfies constraints 19–21. In particular, because of constraint 21, the rows determined by x and claude are different.

Constraint Lingo provides several constructs for representing ordering relations in integer domains. The one used for this puzzle is OFFSET. In the simplest case, it indicates a numeric difference between the values of an integer attribute in two rows. Lines 15 indicates that the value of the position attribute of two rows differs by ± 1.

We omit discussion of other features to Constraint Lingo due to space limitations.

Translation to *smodels*

We use Constraint Lingo not only as a tool to model tabular CSPs; we also execute programs in Constraint Lingo to solve the corresponding tCSPs. We first compile Constraint Lingo programs into the syntax of an answer-set programming formalism. In this paper we describe *smodels* (Niemelä & Simons 1997; 2000) as the target environment. However, we have also targeted other ASP formalisms, such as *dcs* (East & Truszczyński 2000); *dlv* (Eiter *et al.* 1998) could also be used. Once a Constraint Lingo program is compiled into *smodels* format, we use *lparse* and *smodels* to execute it and determine solutions.

We have experimented with several representations of tables in predicates of *smodels*; the speed of execution is quite sensitive to using a good representation. Here we show only one representation.

Each class (attribute) in the Constraint Lingo program is represented in *smodels* as a predicate. Domain elements are atoms defining its extension:

```
person(claude).        person(jeanne).
person(kate).          person(liana).
person(martin).        person(robert).
```

We represent the table by binary predicates (projections onto pairs of attributes), called *cross-class predicates*. For instance, for classes person and soda we introduce a predicate person_soda (we order class names lexicographically). Its first attribute is of type person and the second of type soda. The presence of an atom person_soda(a,b) in a stable model indicates that in the corresponding solution person a drinks soda b.

Given two attributes att1 and att2, each domain value of one is on the same row as is exactly one domain value of the other. This constraint is modeled in *smodels* by two clauses (V1 and V2 stand for variables of types att1 and att2, respectively):

```
1 {att1_att2(V1,V2):att1(V1)} 1 :- att2(V2) .
1 {att1_att2(V1,V2):att2(V2)} 1 :- att1(V1) .
```

Cross-class predicates obey a closure property: If a person is on the same row as a soda, and that soda is on the same row as a particular visit, then that person must also be on the same row as that visit. In general terms, given three attributes att1, att3, we represent this constraint as follows:

```
att1_att3(V1,V3) :-
        att1(V1), att2(V2), att3(V3),
        att1_att2(V1,V2), att2_att3(V2,V3).
```

Partitions are handled similarly to classes, except:

1. Although each class member must be associated in a cross-class predicate with a partition member, not every partition member must be associated with a class member.
2. Although a class member may be associated in a cross-class predicate with only one member of a partition, a partition member may be associated with multiple class members.
3. Transitivity does not apply through partition members.

Some constraint problems underspecify class membership (in other words, the *all_used* constraint does not hold). For example, an attribute month might range over values from January to July, but only five of these seven months may actually participate in solutions. Underspecified classes require no modification to the propositional code we generate.

We must design a translation for each constraint form in Constraint Lingo into the language of *smodels*. In most cases, it is straightforward, although it sometimes requires significant tinkering to find the most efficient translation. We describe translation by discussing several examples pertaining to "French Phrases, Italian Soda" problem.

- If REQUIRED specifies that three terms specify the same row, it gives rise just to three facts. For instance:

  ```
  REQUIRED quebec robert 3
  ```

  Gives rise to the following three facts:

  ```
  person_position(robert,3).
  person_visits(robert,quebec).
  position_visits(3,quebec).
  ```

  Closure allows us to omit one of these facts.
- If a single REQUIRED specifies an alternative (as in line 13 of the example code) we represent it by the smodels clause

  ```
  1 {soda_visits(blueberry,quebec),
     soda_visits(lemon,quebec)} 2.
  ```
- A conflict between two members is represented by a failure rule. We represent CONFLICT quebec 1 by

  ```
  :- position_visits(1,quebec).
  ```

  A single CONFLICT statement in Constraint Lingo can include many class members. Each pair of members generates a failure rule of this form, except for pairs where both members are in the same class.

- The MATCH statement generates two groups of rules. The first group requires that every member of the first list be associated with at least one member of the second list, and vice-versa. For instance, `MATCH 2 4 6, kate claude tangelo` generates a rule requiring that position 2 be associated with one of `kate`, `claude` and `tangelo`:

```
1 {person_position(kate,2),
   person_position(claude,2)
   position_soda(tangelo,2} 1.
```

  The second group introduces a conflict among all members of the first list and a conflict among all members of the second list. The Constraint Lingo `MATCH 2 4 6, kate claude tangelo` has only members of a single class in the first list; these members can never be associated in a single row. But the second list has two classes, so it generates rules to separate persons `kate` and `claude` from soda `tangelo`.

- An ordering statement that specifies a fixed relation between two tuples generates two rules: An implication setting the value of the second row, and a failure rule requiring that that value be in range. For instance, `OFFSET 1 position: lemon haiti` generates

```
position_visits(Position1+1,haiti) :-
    position(Position1),
    position_soda(Position1,lemon).
:-  position(Position1),
    position_soda(Position1,lemon),
    not position(Position1+1).
```

  Similar rules are generated for fixed multiplicative relations.

- An ordering statement that specifies a bounded relation between two rows generates a single failure rule. For instance, `OFFSET >2 position: lemon haiti` generates

```
:-  position(Position1; Position2),
    position_soda(Position1,lemon),
    position_visits(Position2,haiti),
    Position1 + 2 >= Position2.
```

- An ordering statement that specifies a ± offset between two tuples requires an auxiliary predicate indicating that two values differ by that offset. For instance, `OFFSET +-2 position: lemon haiti` introduces a `differ_position_2` predicate defined by two implications:

```
differ_position_2(Position1, Position1+2) :-
    position(Position1).
differ_position_2(Position1+2, Position1) :-
    position(Position1).
```

  The constraint itself is then expressed by a failure rule:

```
:-  position(Position1; Position2),
    position_soda(Position1,lemon),
    position_visits(Position2,haiti),
    not differ_position_2(Position1,Position2).
```

- Ordering constraints with respect to a circular numeric class generate similar rules, but the arithmetic details differ slightly, using modular arithmetic.

- A variable introduced to capture secondary constraints introduces an auxiliary predicate that forces the variable to bind to one of the atoms of some arbitrary class. For instance, `VAR x` generates

```
1 {tuple_x(X): soda(X)} 1.
```

  When a variable participates in a statement, it is represented by a variable constrained by the introduced predicate. For example, `AGREE man: haiti 3 x` generates

```
gender_visits(man,haiti).
gender_position(man,3).
gender_soda(man,X) :- tuple_x(X).
```

- In order to guide the postprocessor and generate the solution table, the translated program includes a summary fact. In our example, this fact is

```
answer(visits).
```

  This fact informs the postprocessor that a solution is a set of rows, each of which links the class `visits` to the other classes and partitions through cross-class predicates.

We use the *lparse* program to ground the resulting propositional program (Niemelä & Simons 1997). This program replaces each rule containing variables with many rules, one for each combination of values that all its variables can have, subject to the type constraints in the rules. The result is called a *grounded* program, because all the variables have been replaced by ground atoms.

We then use the *smodels* program to search for stable models of the grounded program (Niemelä & Simons 1996). A stable model is a collection of grounded predicates that are mutually consistent (they trigger no failure rule) and required (each is implied by at least one implication). Each grounded instance of a cross-class predicate involving the atom specified by the `answer` predicate represents a row. We post-process the output of *smodels* to extract those instances and to format them. One of the solutions of the relaxed example (and the only solution of the original one) is

person	pos	soda	visits	gender
claude	6	tangelo	haiti	man
jeanne	1	grapefruit	ivory	woman
kate	4	kiwi	tahiti	woman
liana	5	peach	belgium	woman
martin	3	lemon	quebec	man
robert	2	blueberry	martinique	man

Each row represents a tuple. For example, the first line says that Claude, sitting at position 6, drinks tangelo soda, plans to visit Haiti, and is a man. The complete version of the french-phrases puzzle has exactly one solution (as is expected for all puzzles in *Dell Logic Puzzles*).

Discussion and Conclusions

We have focused on a novel use of database query languages. Formulas of such languages are normally used for deriving sets of tuples (that is, tables or answers to queries) from the tables stored in the actual database system. These formulas involve predicate symbols referring either to existing database tables or to built-in relations. In our approach, we consider sentences only. They are built of predicates referring to known and built-in relations and, in addition, to some unknown tables. Such a sentence can be viewed as a constraint on tables referred to by these additional predicate symbols. For some tables interpreting them, the sentence is satisfied. Others may fail to satisfy it. Thus the sentences of a database query language, expanded by additional predicate symbols to refer to unknown relations, serve as means to define constraints on tables, not on individual tuples.

The analogy to elementary classes in model theory is clear. There, a collection of sentences serves as a constraint imposed on relational structures. Compared with model theory, we impose, however, two limitations. First, the languages itself must be based on a finite alphabet, with no function symbols, thus with a finite Herbrand base. Second, the relational structures that serve as models are Herbrand models. In view of the limitation discussed above, these models are always finite,

In the paper we studied a formal language to express constraints on tables that is based on tuple relational calculus. This language is not suitable for actual programming. While a logician might find it easy to write formulas in tuple relational calculus, a programmer might not. Thus, we argue that in order to support constraint programming it is necessary to develop high-level front ends supporting modeling of frequently occurring constraints. In this paper we have described a prototype of such a high-level front end, Constraint Lingo, able to express a subclass of tabular CSPs that can be specified by tuple relational calculus. The results presented in this paper demonstrate the feasibility of this approach. We have used Constraint Lingo to model over 80 constraint problems describing logic puzzles and some combinatorial problems such as graph coloring. In the future, we intend to extend Constraint Lingo to the entire language of tuple relational calculus.

A main advantage of our approach is that Constraint Lingo programs can be automatically translated into ASP formalisms. In this paper, we have presented a translation into the syntax of *smodels*. Consequently, *smodels* can be used to solve constraint problems expressed in the language of Constraint Lingo. This approach seems to be computationally practical. In our computational experiments, even hard (four-star) logic puzzles, whose ground versions consist of thousands of rules, are processed quickly, never requiring more than 10 seconds (for our best representation).

ASP systems other than *smodels* can be used as back ends. We have experimented with *dcs* and obtained similar results to those obtained with *smodels*. However, more experiments with other back ends are needed. In particular, recent progress in satisfiability checkers (Gent, van Maaren, & Walsh 2000) points to the potential of propositional logic.

We have found that how we represent of Constraint Lingo in ASP formalisms such as *smodels* critically affects efficiency. We continue to develop new representations. The fact that massive improvements are possible underscores our belief that efficient solution of constraint-satisfaction problems depends on a carefully designed front end; even expert programmers in DATALOG$^\neg$ are unlikely to achieve efficient programs without enormous effort.

References

East, D., and Truszczyński, M. 2000. Datalog with constraints. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, 163–168.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. A kr system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the Sixth International Conference on Knowledge Representation and Reasoning (KR '98)*, 406–417. Morgan Kaufmann.

Gent, I.; van Maaren, H.; and Walsh, T. 2000. *SAT2000: Highlights of satisfiability research in the year 2000*. IOS Press.

Maier, D. 1983. *The Theory of Relational Databases*. Computer Science Press.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In Apt, K.; Marek, W.; Truszczyński, M.; and Warren, D., eds., *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 375–398.

Niemelä, I., and Simons, P. 1996. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96*. MIT Press.

Niemelä, I., and Simons, P. 1997. Smodels — an implementation of the stable model and well-foundd semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning (the 4th International Conference, Dagstuhl, Germany, 1997)*, volume 1265 of *Lecture Notes in Computer Science*, 420–429. Springer-Verlag.

Niemelä, I., and Simons, P. 2000. Extending the smodels system with cardinality and weight constraints. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers. to appear.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Ullman, J. 1988. *Principles of Database and Knowledge-Base Systems*. Rockville, MD: Computer Science Press.