SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. maybe later; **00**:1–25 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Constraint Lingo: Towards high-level constraint programming

Raphael Finkel*, Victor W. Marek and Mirosław Truszczyński

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA,
{raphael,marek,mirek}@cs.uky.edu

SUMMARY

Logic programming requires that the programmer convert a problem into a set of constraints based on predicates. Choosing the predicates and introducing appropriate constraints can be intricate and error-prone. If the problem domain is structured enough, we can let the programmer express the problem in terms of more abstract, higher-level constraints. A compiler can then convert the higher-level program into a logic-programming formalism. The compiler writer can experiment with alternative low-level representations of the higher-level constraints in order to achieve a high-quality translation. The programmer can then take advantage of both a reduction in complexity and an improvement in runtime speed for all problems within the domain.

We apply this analysis to the domain of tabular constraint-satisfaction problems. Examples of such problems include logic puzzles solvable on a hatch grid and combinatorial problems such as graph coloring and independent sets. The proper abstractions for these problems are rows, columns, entries, and their interactions.

We present a higher-level language, Constraint Lingo, dedicated to problems in this domain. We also describe how we translate programs from Constraint Lingo into lower-level logic formalisms such as the logic of propositional schemata. These translations require that we choose among competing lower-level representations in order to produce efficient results.

The overall effectiveness of our approach depends on the appropriateness of Constraint Lingo, our ability to translate Constraint Lingo programs into high-quality representations in logic formalisms, and the efficiency with which logic engines can compute answer sets.

We comment on our computational experience with these tools in solving both graph problems and logic puzzles.

KEY WORDS: logic puzzles, logic programming, constraint satisfaction, tabular constraint satisfaction

Contract/grant sponsor: National Science Foundation; contract/grant number: 9619233, 9874764, 0097278

Received February 2003

Copyright © maybe later John Wiley & Sons, Ltd.

1. Introduction

Logic programming was introduced in the mid 1970s as a way to facilitate computational problem solving and software development [1]. The idea was to regard logic theories as programs and formulas as representations of computational tasks, and to apply automated reasoning techniques, most notably, resolution with unification, as the computational mechanism. Researchers expected that logic programming would quickly become a dominant programming paradigm because of its declarative nature: it allows programmers to focus on modeling problem specifications in a declarative way as theories and frees them from the need to describe control. These expectations were reinforced by the emergence of Prolog [2]. However, despite the initial excitement generated by logic programming and its prominent role in the fifth-generation computing initiative in Japan, logic programming has been slow in winning broad acceptance and has yet to live up to early expectations.

This paper presents our attempt to address this problem. Logic programming requires the programmer to cast a problem into the language of predicates and their interrelations, a task that is often intricate and error-prone. It is more productive to program with domain-appropriate abstractions that are automatically compiled into efficient and correct low-level logic programs.

We demonstrate this thesis in the restricted domain of constraint-satisfaction problems whose solutions have the structure of a table. In this paper, we describe the Constraint Lingo language for expressing these tabular constraint-satisfaction problems. We show how to translate programs in this language into a variety of lower-level logic programs that can be run on standard logic engines.

Low-level approaches to constraint-satisfaction problems have been investigated for several years. First, constraint-logic programming [3] has been used with great success. Solvers such as ECL^{i}PS^{e} [4] can be used to represent and solve such problems. Second, recent research has modeled constraint-satisfaction problems as DATALOG$^{\neg}$ programs for which stable models represent solutions [5, 6]. Programs such as *smodels* [7] compute stable models [8] of such programs. Third, constraint-satisfaction problems can be modeled as disjunctive logic programs; *dlv* [9] can compute answer sets of those programs. Fourth, the logic of propositional schemata forms an answer-set programming formalism that can be used for solving constraint-satisfaction problems [10]. Fifth, optimization programming solvers such as OPL [11] deal primarily with techniques such as linear and integer programming, but also incorporate constraint programming and scheduling. Together, we call programs that compute solutions to logic programs in any of these formalisms "logic engines", even though solvers in the fifth class can be based on C++ or Java, and do not present a predicate-logic view to programmers.

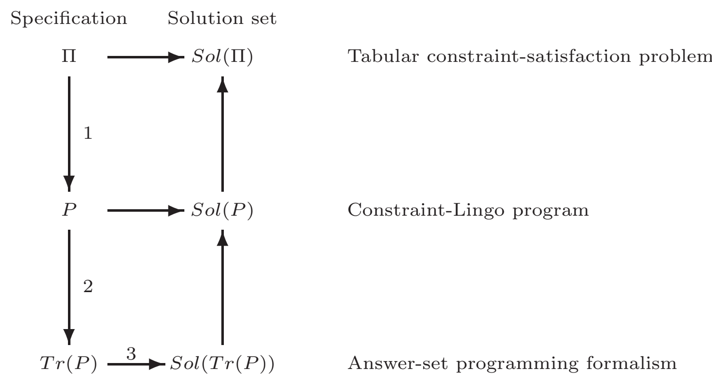
While these logic-based formalisms for specifying constraints are expressive, they all suffer from the fact that they are awkward even for experienced programmers and logicians to use. The problem is that the connectives offered by logic do not correspond well to high-level constraints occurring in actual problems, even when connectives include the extended syntax implemented by *smodels* or by ECL^{i}PS^{e}.

Solving a constraint-satisfaction problem should be a three-step process. (1) Represent a statement of the problem (often given informally as free text) in some high-level modeling language. We refer to this step as *modeling* or *programming*. (2) Translate this representation

Copyright © maybe later John Wiley & Sons, Ltd.

into a target formalism for which good automated reasoning techniques are available. We refer to this step as *compilation*; it is usually fully automated. (3) Apply automated reasoning techniques to the compiled representation in order to construct solutions to the original problem if they exist. We refer to this step as *computation*.

The following figure summarizes the flow of solving constraint-satisfaction problems and introduces some of our notation. A programmer represents problem Π as program P. An automatic translator converts P into $Tr(P)$. A logic engine computes the solution set $Sol(Tr(P))$ for $Tr(P)$. Each model $M \in Sol(Tr(P))$ represents a solution to the Constraint Lingo program and hence to the original problem.



This three-step approach is not unique to constraint satisfaction; it is quite common in all computational areas. (1) Using programming languages to solve problems follows the same general pattern of programming, compiling, and computing. (2) To retrieve information from a database, we first write a query in some query language (programming). This query is then analyzed, optimized, and transformed into a computational plan, such as an expression in relational algebra. Finally, this plan is executed and the answer is computed. (3) A concrete example of the use of this approach in AI is propositional satisfiability planning [12]. In the BlackBox approach [13], to solve a planning problem we first build its formal representation in a high-level planning language such as STRIPS [14, 15] or PDDL [16], then compile it into a propositional CNF theory, and finally solve the original planning problem by using these propositional satisfiability programs to find models of the compiled theory.

From this perspective, due to its limited repertoire of means to express constraints, logic formalisms should rather be viewed as low-level constraint-modeling languages. In order to use them for solving constraint problems, one needs a high-level modeling formalism tailored to the actual problems, coupled with techniques to translate theories in this high-level formalism into logic programs. An expressive language for representing constraints should facilitate programming, and good compilation techniques should result in code amenable to efficient processing by any logic engine.

In this paper we present a new constraint-modeling language, Constraint Lingo, well suited for modeling tabular constraint-satisfaction problems. To demonstrate its utility, we show (1) how to encode logic puzzles and several graph problems in Constraint Lingo, (2) how to

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

compile Constraint Lingo programs into several logic formalisms, and (3) how well logic engines compute answer sets for the compiled programs.

Our experience with Constraint Lingo (the current implementation, problem suite, and documentation is available [17]) supports our thesis. Although we find it hard to program constraint-satisfaction problems directly in logic formalism, we find that (1) it is quite easy (and even fun) to program these problems in Constraint Lingo, (2) compilation is completely automated, and (3) the resulting programs are efficient to run.

This paper makes three contributions:

1. It proposes a technique for using logic formalisms as computational tools. We contend that logic formalisms should preferably be used as computational back-ends accompanying a more user-friendly high-level programming language. Programming ought to be done in this higher-level language; programs need to be compiled to low-level representations and then processed.
2. We illustrate our proposal by developing a specific language for modeling constraint problems. We also illustrate compilers into several computational logic-based back-ends and demonstrate the viability of our approach.
3. Our approach opens interesting research directions for constraint-satisfaction programming:
    - design of high-level languages for logic-programming application areas,
    - design of compilers and their optimizations,
    - design of software-development tools.

This paper is organized as follows. We present tabular constraint-satisfaction problems and a particular logic puzzle in Section 2. We introduce the syntax of Constraint Lingo and its semantics in Section 3, applying it to a specific logic puzzle. We apply Constraint Lingo to graph problems in Section 4. We show how Constraint Lingo is translated into *smodels* in Section 5 and show some compiler optimizations for that translation in Section 6. We show how compiled code differs from *smodels* for other logic engines, in particular, *dlv* in Section 7.1 and ECL$^{\text{i}}$PS$^{\text{e}}$ in Section 7.2. We present results of timing studies in Section 8 and final remarks in Section 9.

2. Tabular constraint-satisfaction problems

The Constraint Lingo language is tuned to *tabular constraint-satisfaction problems* (**tCSP**s), in which it is convenient to think about solutions as having a 2-dimensional array structure. Such problems specify columns in the tables by assigning them names and by indicating the domain of each column, that is, the set of elements that can appear in the column. They also specify the number of rows. Further constraints typically relate entries in a single row or column, but more complex constraints are also possible.

An **attribute** means a pair (a, D_a), where a is the **name** of the attribute and D_a is its **domain**, a nonempty set of elements. For our purposes, all attribute domains are finite. We commonly refer to an attribute by its name. A **table schema** is a sequence of attributes with distinct names.

Copyright © maybe later John Wiley & Sons, Ltd.

Let $\mathcal{S} = \langle a_1, \ldots, a_n \rangle$ be a table schema. We call any subset $T \subseteq D_{a_1} \times \ldots \times D_{a_n}$ a **table in schema** $\mathcal{S}$. We use the term **table** rather than **relation**, the standard term for a subset of a Cartesian product, to emphasize intuitions arising in the context of tCSPs. In particular, we regard a table as a two-dimensional structure consisting of all its tuples written sequentially as **rows**. Likewise, a **column** is the sequence of elements from the domain of an attribute appearing in the appropriate position in all rows of the table. We denote the set of all tables in $\mathcal{S}$ by $\mathit{Tab}(\mathcal{S})$.

A **constraint** on tables in $\mathcal{S}$ is any subset of $\mathit{Tab}(\mathcal{S})$*. We say a table **satisfies** a constraint if it is a member of the subset. A **tabular constraint satisfaction problem (tCSP)** consists of a table schema $\mathcal{S}$ and a collection $\mathcal{C}$ of constraints on tables in $\mathit{Tab}(\mathcal{S})$. Given a tCSP Π, the set of solutions to Π consists of all those tables in $\mathit{Tab}(\mathcal{S})$ that satisfy all the constraints in $\mathcal{C}$. We denote this set as $\mathit{Sol}(\Pi) = \bigcap_{c \in \mathcal{C}} c$.

The most common table constraints are **all-different** and **all-used**. A table $T \in \mathit{Tab}(\mathcal{S})$ satisfies the **all-different** property with respect to the attribute $a \in \mathcal{S}$ if no element of D_a appears more than once in the column a of T. A table $T \in \mathit{Tab}(\mathcal{S})$ satisfies the **all-used** property with respect to the attribute $a \in \mathcal{S}$ if each element of D_a appears at least once in the column a of T. We say that an attribute $a \in \mathcal{S}$ is a **key** for a table T in $\mathcal{S}$ if T satisfies both the all-different and all-used constraints† with respect to a.

We are only interested in tCSPs with at least one key attribute. Without loss of generality, we assume that the first attribute in the schema, say a_1, is so distinguished. This requirement implies that the number of rows in solution tables is the cardinality of D_{a_1}.

This assumption is motivated by the following considerations. First, it is often satisfied by problems appearing in practice, in particular, by the puzzle and graph problems discussed in this paper. Second, general constraint-satisfaction problems assume a fixed set of variables. In tCSPs, variables whose values need to be established correspond to individual table entries. The schema determines the number of columns. In order to fix the number of variables of a tCSP, we have to fix the number of rows. Designating an attribute as a key is one way of doing so. Third, a class of tables satisfying our assumption can be uniquely decomposed into collections of 2-column projections on pairs of attributes. This property has implications for translations of Constraint Lingo programs into low-level logic formalisms. We discuss this matter in more detail in Section 5.

Scheduling problems are examples of tCSPs, with each row in a solution table representing a single item of the schedule (such as time, location, resources needed). Graph problems can also often be cast as tCSPs. For instance, a solution to a graph-coloring problem is a table consisting of two columns, one for vertices and the other for colors. The rows in the table specify the assignment of colors to vertices.

*Usually such a constraint is not given explicitly as a set of relations but rather as a formula in some language.
†We differ here from database terminology, in which only the all-different constraint is required for an attribute to be a key.

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

Logic puzzles are good examples of tCSPs. Throughout this paper, we use the "French Phrases, Italian Soda" puzzle (or **French puzzle**, for short)[‡] as a running example to illustrate the syntax and the semantics of Constraint Lingo:

> 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 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.
4. The three people who sat in even-numbered seats are Kate, Claude, and a person who didn't drink lemon soda, in some order.

This puzzle can be viewed as a tCSP with the schema consisting of five attributes: `name`, `gender`, `position`, `soda` and `country` (each with its associated domain). The space of possible solutions to this puzzle is given by the set of tables whose rows describe people and whose columns describe relevant attributes. The attributes `name`, `position`, `soda` and `country` are implicitly required to be key, but `gender` is not (it does not satisfy the all-different property). There is only one solution satisfying all nine clues of the French puzzle:

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

3. Syntax and Semantics of Constraint Lingo

The syntax of Constraint Lingo is line-oriented. Every non-empty line of Constraint Lingo constitutes a *declaration* or a *constraint*. For better readability, declarations usually precede constraints, but Constraint Lingo only requires that every atom be declared before use. Comments are prefixed with the # character.

[‡]Copyright 1999, Dell Magazines; quoted by permission. We present only four of the nine clues.

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

The goal of a Constraint Lingo program P is to specify a tCSP Π. Declarations of the program P describe the schema of the problem Π and impose all-different and all-used constraints. Constraints of the program P describe all other constraints of the problem Π. By specifying a tCSP, a Constraint Lingo program P can be regarded as a representation of all tables in $Sol(\Pi)$.

To describe the set of tables that are solutions to a Constraint Lingo program P we proceed as follows. We first specify the set of tables that is determined by B, the declaration part of P. We then describe, for each constraint C in the rest of P, which of the tables specified by the B satisfy it. Those tables that satisfy all constraints constitute solutions to P.

3.1. Declarations

Two different types of attributes (columns) can be declared in Constraint Lingo.

- `CLASS` *classname*: $member_1$ $member_2 \dots member_k$

This syntax declares a **class attribute** (column) with the name *classname* and the domain consisting of elements $member_1$, $member_2$, $\dots$, $member_k$. Classes are columns in which every element is different.

For the French puzzle, for example, we have

```
CLASS name: claude jeanne kate liana martin robert
CLASS soda: blueberry lemon peach tangelo kiwi grapefruit
CLASS visits: quebec tahiti haiti martinique belgium ivory
```

If the domain elements are all integers (our parser only allows nonnegative integers) in a range from *first* to *last*, we may specify the class by writing:

`CLASS` *classname*: *first* `..` *last* `[circular]`

In the French puzzle, we write

```
CLASS position: 1 .. 6 circular
```

The optional `circular` keyword indicates that the range is intended to be treated with modular arithmetic so that $last + 1 = first$. We refer to classes all of whose members are numeric as **numeric** classes; the others are **list** classes.

- `PARTITION` *partitionname*: $member_1$ $member_2 \dots member_k$

This syntax declares a **partition attribute** (column) with the name *partitionname* and the domain consisting of elements $member_1, member_2, \dots, member_k$. Members of a partition attribute may occur any number of times (even 0) in their column.

In the French puzzle, we write

```
PARTITION gender: men women
```

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

We require that all class and partition names be distinct. We also require at least one list class (so we can be sure how many rows there are) and that the domains of all list-class attributes be of the same cardinality. This requirement corresponds to the restriction we impose on tCSPs that at least one attribute must be key. However, we often find it useful to let numeric classes include values that turn out not to appear in the solution. We therefore let numeric classes have more values than other classes.

Each attribute constitutes a disjoint domain of elements. If we need the same element (such as a number) in two attribute domains, we disambiguate the domains in the constraint part of the program by **qualifying** the element: *attributename.element*.

Let B be the declarations of a Constraint Lingo program P. These declarations define a table schema, say $\mathcal{S}_B$, which consists of all class and partition attributes. In addition, B imposes all-different and all-used constraints by designating some attributes as list- and numeric-class attributes. Specifically, B restricts the space of tables in $Tab(\mathcal{S}_B)$ to those that satisfy the all-different constraint with respect to class attributes and also the all-used constraint with respect to list-class attributes. Neither restriction applies to partition attributes. We denote this set of tables, which constitute the solution space, as $SS(B)$. We regard each table in this set as a model of declarations in B and view the set $SS(B)$ as providing the semantics for B.

3.2. Constraints

The schema defined by declarations B introduces identifiers (such as class names and domain members) that are then used in the constraints found in the rest of the Constraint Lingo program. We discuss this syntax now. For each constraint we introduce, we define its semantics in the terms of tables in the set $SS(B)$.

3.2.1. Rownames

Constraints often concern properties of table columns and rows. To refer to columns we use their class or partition names. To refer to a row we use a **rowname**. A rowname may be any element of a class domain, which uniquely refers to one row because of the all-different constraint and the disjoint nature of domain elements (ensured if necessary by qualifying them). In addition, we may introduce a variable as a rowname:

- `VAR` *variablename*

Variables must be distinct from each other and from all domain elements to avoid ambiguity.

We now give the syntax and semantics of the constraints in Constraint Lingo given the set of declarations B. When describing the semantics we assume for now that constraints do not involve variables. We later lift this assumption.

3.2.2. `REQUIRED` *and* `CONFLICT`

- `REQUIRED` $rowname_1$ $rowname_2$ ...

Copyright © maybe later John Wiley & Sons, Ltd.

A table from $SS(B)$ satisfies a `REQUIRED` constraint if the given *rowname*s specify the same row, that is, if they appear in the same row of the table.

We would encode a clue "The person traveling to Quebec drank blueberry soda" as

```
REQUIRED quebec blueberry
```

- `REQUIRED` $rowname_1$ $rowname_2$ `[OR | XOR | IFF]` $rowname_3$ $rowname_4$

This embellished `REQUIRED` constraint is satisfied only by those tables from $SS(B)$ in which $rowname_1$ and $rowname_2$ specify the same row {or, xor, iff} $rowname_3$ and $rowname_4$ specify the same row, depending on the connector used. This constraint gives the Constraint Lingo programmer a limited amount of propositional logic. The effect of the `AND` connective is achieved by writing separate constraints, so we do not include it in Constraint Lingo.

For the French puzzle, we encode part of the first clue as

```
REQUIRED quebec blueberry OR quebec lemon
```

- `CONFLICT` $rowname_1$ ... `[`$partitionelement_1$ ...`]`

The `CONFLICT` constraint excludes those tables in $SS(B)$ in which any two of the given *rowname*s specify the same row. If *partitionelement*s are specified, the constraint also disallows tables in which those *partitionelement*s are found in any rows specified by the given *rowname*s.

We partially encode the first French puzzle clue as:

```
CONFLICT quebec 1
```

We could use a partition element, for example, to stipulate that neither the person drinking kiwi soda nor the person going to Belgium is a man:

```
CONFLICT kiwi belgium men
```

We find that we use `REQUIRED` and `CONFLICT` most heavily. We now turn to less-frequently used constraint types.

3.2.3. Other constraint types

- `AGREE` *partitionelement*`:` $rowname_1$ ...

The `AGREE` constraint is satisfied by those tables in $SS(B)$ in which the rows specified by the given *rowname*s have the given *partitionelement* in the column associated with *partitionelement*.

We use `AGREE` to indicate the genders of the six people:

```
AGREE men: claude martin robert
AGREE women: jeanne kate liana
```

We also use `AGREE` along with `VAR` and `CONFLICT` for clue 3:

Copyright © maybe later John Wiley & Sons, Ltd.

```
VAR brother
CONFLICT brother claude
AGREE men: haiti 3 brother
CONFLICT haiti 3 brother
```

- `DIFFER` *partitionname*`:` $rowname_1$ $rowname_2$ ...

This constraint allows only those tables in $SS(B)$ in which the rows specified by the given *rownames* have different elements in the column associated with *partitionname*.

For example, we could have a clue stating that the person visiting the Ivory Coast and the one drinking blueberry soda are of different genders; we would encode that clue as:

```
DIFFER gender: ivory blueberry
```

- `SAME` *partitionname*`:` $rowname_1$ $rowname_2$ ...

This constraint allows only those tables in $SS(B)$ in which the rows specified by the given *rownames* have the same elements in the column associated with *partitionname*.

To represent, for instance, that the person visiting the Ivory Coast has the same gender as the one drinking kiwi soda, we would write:

```
SAME gender: ivory kiwi
```

- `USED` *element*

This constraint disallows all those tables in $SS(B)$ in which the given *element* does not appear in its associated column. We employ this constraint to force a particular partition element or numeric-class element to be used in a solution.

The French puzzle tells us who are the men and who are the women, but if it only told us that there is at least one man, we would encode that clue as:

```
USED men
```

- `USED` n `<=` *partitionelement* `<=` m

In this constraint, n and m must be nonnegative integers. Either the n `<=` or the `<=` m or both may be absent. The default value for n is 1, and the default value of m is ∞. This constraint allows only those tables in $SS(B)$ where the *partitionelement* appears (in its associated column) a number of times k such that $n \leq k \leq m$.

To encode a clue telling us there are at least 2 but not more than 3 women, we would write:

```
USED 2 <= women <= 3
```

- `MATCH` $rowname_1$ ... $rowname_k$ `,` $rowname'_1$ ... $rowname'_k$

This constraint allows only those tables in $SS(B)$ in which: (1) all the rows specified by the first set of *rownames* are distinct, (2) all the rows specified by the second set of *rownames* are distinct, (3) those two sets of rows are identical.

We encode the fourth clue by a combination of `MATCH` and `VAR`:

Copyright © maybe later John Wiley & Sons, Ltd.

```
VAR unlemon
MATCH 2 4 6, kate claude unlemon
CONFLICT unlemon lemon
```

• `BEFORE` *classname*: *rowname*$_1$ *rowname*$_2$
The given *classname* must be a non-circular numeric class. Let v_1 and v_2 be the elements in the column specified by *classname* and in the rows specified by *rowname*$_1$ and *rowname*$_2$, respectively. The constraint allows only those tables in $SS(B)$ in which $v_1 < v_2$.

We cannot use `BEFORE` in the French puzzle, because a circular numeric class has no order. Ignoring circularity, we could indicate that Jeanne is sitting in an earlier-numbered position than the person going to Haiti by saying:

```
BEFORE position: jeanne haiti
```

• `OFFSET [ + | * | +- | > | ! | !+- ]` n *classname*: *rowname*$_1$ *rowname*$_2$
The given *classname* must again be a non-circular numeric class. Let v_1 and v_2 be the elements in the column specified by *classname* and in the rows specified by *rowname*$_1$ and *rowname*$_2$, respectively. The six variants of this constraint allow only such tables in $SS(B)$ where $v_1+n = v_2$, $v_1*n = v_2$, $v_1 \pm n = v_2$, $v_1+n > v_2$, $v_1+n \neq v_2$, or $v_1 \pm n \neq v_2$, respectively.

Again, `OFFSET` makes no sense in the French puzzle because `position` is a circular numeric class, but ignoring circularity, we could say that Kate is sitting in a position twice as large as Robert's by saying:

```
OFFSET *2 position: robert kate
```

3.2.4. Variables

We have used variables intuitively in some of our examples above; we now extend the description of Constraint Lingo semantics when variables appear in constraints. Let P be a Constraint Lingo program with variables $x_1, \ldots, x_k$. We say that a table T, of the type specified by the declarations of P, satisfies P if there is a list class C and elements $v_1, \ldots, v_k$ (not necessarily distinct) from the domain of this class such that the table T satisfies the Constraint Lingo program obtained by removing all variable declaration statements from P and by instantiating in P every occurrence of x_i with v_i. In other words, we can associate each variable with some row, represented by a value in a list class. In the French puzzle, the variables `unlemon` and `brother` used in the examples above both turn out to be associated with Robert; we could call that row `robert`, `blueberry`, or `martinique`, depending what list class we wish to use as our class C.

3.2.5. French puzzle

A complete translation of the French puzzle clues is as follows.

Copyright © maybe later John Wiley & Sons, Ltd.

```
#1
    REQUIRED quebec blueberry OR quebec lemon
    CONFLICT quebec 1
$2
    OFFSET !+-1 position: robert kate
    OFFSET 3 position: peach
#3
    VAR brother
    AGREE men: haiti 3 brother
    CONFLICT brother claude
#4
    VAR unlemon
    MATCH 2 4 6, kate claude unlemon
    CONFLICT unlemon lemon
```

3.2.6. Solutions

Let P be a Constraint Lingo program and let B denote all declarations in P. A table $T \in SS(B)$ is a **solution** to P if it satisfies all constraints in P. We denote the set of all solution tables for a Constraint Lingo program P by $Sol(P)$. A Constraint Lingo program **encodes** a tCSP problem Π if $Sol(P) = Sol(\Pi)$.

Let P be a finite Constraint Lingo program with the declaration component B. One can show, based on our discussion above, that given a table $T \in SS(B)$, checking whether T satisfies all constraints in P can be accomplished in time polynomial in the size of P and T. However, the tables in the set $SS(B)$ have dimensions that are polynomial in the size of B (and so, in the size of P). It follows that deciding whether a finite Constraint Lingo program has solutions is in the class NP. In the next section, we show a polynomial reduction of the graph 3-colorability problem to that of deciding whether a Constraint Lingo program has solutions. The problem to decide whether a finite Constraint Lingo program has solutions is therefore NP-complete.

4. Applying Constraint Lingo to graph problems

Despite a restricted repertoire of operators aimed initially at solving logic problems, Constraint Lingo is sufficient to model such important combinatorial problems as independent sets, graph coloring, and finding Hamiltonian cycles.

An *independent set* in a graph is a set of v vertices no two of which share an edge. The independent-set problem is to find an independent set with at least k vertices. We represent the problem in the following Constraint Lingo program, setting, for concreteness, $v = 100$ and $k = 30$, with edges (2,5) and (54,97). There are two attributes: a class `vertex`, to represent vertices of the graph (line 1 below) and a partition `status`, to indicate the membership of each vertex in an independent set (line 2). We employ `USED` to constrain the independent set to have at least k elements (line 3). The `REQUIRED` constraints in lines 4 and 5 enforce the independent-set constraint.

```
1  CLASS vertex: 1..100 # v = 100
```

Copyright © maybe later John Wiley & Sons, Ltd.

```
2  PARTITION status: in out
3  USED 30 <= in # k = 30
4  REQUIRED 2 out OR 5 out # edge (2,5): at least one vertex is out
5  REQUIRED 54 out OR 97 out # edge (54,97): at least one vertex is out
```

The k-graph-coloring problem is to find an assignment of k colors to vertices such that vertices sharing an edge are assigned different colors. We use two attributes, `vertex` and `color`, to define the set of vertices and the colors to use. The following Constraint Lingo program encodes the 3-coloring problem for the same graph as before. We enforce the coloring condition by means of `DIFFER` constraints (lines 3 and 4). We use qualified notation in lines 3 and 4 to disambiguate `vertex.2` from `color.2`. The other numbers in the program are already unambiguous, but qualified notation improves clarity.

```
1  CLASS vertex: 1..100
2  PARTITION color: 1..3 # looking for 3-coloring
3  DIFFER color: vertex.2 vertex.5 # edge (2,5)
4  DIFFER color: vertex.54 vertex.97 # edge (54,97)
```

The Hamiltonian-cycle problem is to enumerate, without repetition, all the vertices of an undirected graph in an order such that adjacent vertices in the list share an edge, as do the first and last vertices in the list. We use two numeric attributes: `vertex` and `index`. We enforce the Hamiltonicity condition using the construct `OFFSET`: For every edge not in the graph, the positions of its end vertices in the enumeration must not be consecutive integers (with the last and the first vertices also regarded as consecutive). For a specific example, let us consider a graph missing only two edges: (2,5) and (54,97). The corresponding Constraint Lingo program follows.

```
1   CLASS vertex: 1..100
2   CLASS index: 1..100 circular
3   OFFSET !+-1 index: vertex.2 vertex.5 # no edge (2,5)
4   OFFSET !+-1 index: vertex.54 vertex.97 # no edge (54,97)
```

Other combinatorial problems can often be posed in a similar fashion in Constraint Lingo.

5. Translation of Constraint Lingo into *smodels*

We demonstrate compiling Constraint Lingo programs into the formalism of *smodels* [7], that is, we construct an *smodels* program $Tr(P)$. All our code for translating Constraint Lingo, along with over 150 Constraint Lingo programs, is available to the interested reader [17].

Smodels is an extension of logic programming with negation with the semantics of stable models. We assume that the reader is familiar both with the syntax of *smodels* and with its semantics.

Following our earlier discussion, solutions to a Constraint Lingo program P are tables in the schema defined by P. The set of all tables determined by the declaration part B of P is denoted by $SS(B)$. To capture the semantics of P, we need to represent tables from $SS(P)$.

Copyright © maybe later John Wiley & Sons, Ltd.

Tables with n columns correspond naturally to n-ary predicates, so a straightforward approach is to use an n-ary predicate symbol, say *sol*, and to design $Tr(P)$ so that extensions of *sol* in stable models of $Tr(P)$ correspond precisely to tables in $SS(P)$.

Although straightforward, this approach has a disadvantage. The arity of *sol* is the number of attributes of Π, which can be very high. *Smodels* programs involving predicates of high arity lead to ground programs whose size makes processing impractical. Happily, n-ary tables can be represented by collections of their two-column subtables if the tables have at least one key attribute. Tables specified by Constraint Lingo programs fall into this category. We take advantage of this representation to design the translation $Tr(P)$. We now describe this translation; for reasons of space, we omit formal statements of its key properties and outlines of correctness proofs.

Let P be a Constraint Lingo program with declarations B. We assume that B specifies a schema $\mathcal{S} = (a_1, \ldots, a_n)$ where, for some $1 \leq \ell \leq k \leq n$, $a_1, \ldots, a_\ell$ are list-class attributes, $a_{\ell+1}, \ldots, a_k$ are numeric-class attributes, and the remaining $a_{k+1}, \ldots, a_n$ are partition attributes. In particular, a_1 is a list-class attribute.

We now specify the translation $Tr(P)$. The language of $Tr(P)$ is given by (1) the constants forming the domains of attributes of the schema $\mathcal{S}$, (2) the predicate symbols dom_{a_i}, $1 \leq i \leq n$, and (3) the predicate symbols $cross_{a_i,a_j}$, $1 \leq i \leq k$ and $i < j \leq n$. The predicate symbols dom_{a_i} represent attribute domains in $Tr(P)$, and the predicate symbols $cross_{a_i,a_j}$ represent two-column subtables of the solution table (which together determine the solution table). In the case of the French puzzle, for example, the domain predicates include `name(·)` and `soda(·)`; the cross-class predicates include `name_soda(·,·)` and `visits_position(·,·)`.

(1) For every class and partition attribute $a \in P$ we introduce the corresponding predicate dom_a and include in $Tr(P)$ facts $dom_a(v)$ for every element v from the domain of a as described by P. For example, we include the fact `name(claude)`.

(2) For every two list-class attributes a_i, a_j, $1 \leq i < j \leq \ell$, we include in the program $Tr(P)$ the following rules:

$$1\{cross_{a_i,a_j}(V_i, V_j) : dom_{a_j}(V_j)\}1 \text{ :- } dom_{a_i}(V_i).$$
$$1\{cross_{a_i,a_j}(V_i, V_j) : dom_{a_i}(V_i)\}1 \text{ :- } dom_{a_j}(V_j).$$

Informally, the first of these two rules states that for every element v_i of the domain of a_i there is exactly one element v_j from the domain of a_j such that $cross_{a_i,a_j}(v_i, v_j)$ holds (belongs to a stable model). The second rule states the symmetric constraint.

For instance, we have

```
1 {visits_position(Visits,X):visits(Visits)} 1 :- position(X) .
```

This rule means that given a position (such as 2), there is at least and at most 1 location (it turns out to be Martinique) such that the person in that position (it turns out to be Robert) plans to visit that location.

(3) For every list-class attribute a_i, $1 \leq i \leq \ell$, and numeric-class attribute a_j, $\ell+1 \leq j \leq k$, we include in the program $Tr(P)$ the following rules:

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

SOFTWARE PRACTICE & EXPERIENCE

$$1\{cross_{a_i,a_j}(V_i,V_j) : dom_{a_j}(V_j)\}1 \text{ :- } dom_{a_i}(V_i).$$
$$0\{cross_{a_i,a_j}(V_i,V_j) : dom_{a_i}(V_i)\}1 \text{ :- } dom_{a_j}(V_j).$$

The first of these two rules states that for every element v_i of the domain of a_i there is exactly one element v_j from the domain of a_j such that $cross_{a_i,a_j}(v_i, v_j)$ holds (belongs to a stable model). The second rule states that for every element v_j of the domain of a_j there is at most one element v_i from the domain of a_i such that $cross_{a_i,a_j}(v_i, v_j)$ holds (belongs to a stable model). This requirement is weaker then the previous one, a result of the fact that a_j is a numeric-class attribute. We do not require that every element of a numeric-class domain have a match in the domain of a_i in $cross_{a_i,a_j}$, but we still need to require that no element has more than one match.

(4) For every two numeric-class attributes a_i, a_j, $\ell+1 \leq i < j \leq k$ we include in the program $Tr(P)$ the following rules:

$$0\{cross_{a_i,a_j}(V_i,V_j) : dom_{a_j}(V_j)\}1 \text{ :- } dom_{a_i}(V_i).$$
$$0\{cross_{a_i,a_j}(V_i,V_j) : dom_{a_i}(V_i)\}1 \text{ :- } dom_{a_j}(V_j).$$

Informally, these clauses enforce the all-different constraint for a_i and a_j in the two-column table represented by $cross_{a_i,a_j}$ (the only constraint required of numeric-class attributes).

(5) For every list-class attribute a and every partition attribute p, we need to guarantee that atoms of the form $cross_{a,p}(v_a, v_p)$ define a function (not necessarily a bijection) that maps elements of the domain of a to elements from the domain of p. The following rule embodies this guarantee.

$$1\{cross_{a,p}(A,P) : dom_p(P)\}1 \text{ :- } dom_a(A).$$

(6) For every numeric-class attribute a and every partition attribute p, the atoms $cross_{a,p}(v_a, v_p)$ need only define a partial function. We include in $Tr(P)$ the clause:

$$0\{cross_{a,p}(A,P) : dom_p(P)\}1 \text{ :- } dom_a(A).$$

(7) Not every collection of two-column tables can be consistently combined into a single table. In order to achieve consistency, we enforce a transitivity property. For every three class attributes a_h, a_i and a_j, $1 \leq h < i < j \leq k$, we include in $Tr(P)$ the rules:

$$cross_{a_h,a_i}(V_h,V_i) \text{ :- } cross_{a_h,a_j}(V_h,V_j), cross_{a_i,a_j}(V_i,V_j),$$
$$\quad dom_{a_h}(V_h), dom_{a_j}(V_j), dom_{a_j}(V_j).$$
$$cross_{a_h,a_j}(V_j,V_j) \text{ :- } cross_{a_h,a_i}(V_h,V_i), cross_{a_i,a_j}(V_i,V_j),$$
$$\quad dom_{a_h}(V_h), dom_{a_j}(V_j), dom_{a_j}(V_j).$$
$$cross_{a_i,a_j}(V_h,V_j) \text{ :- } cross_{a_h,a_i}(V_h,V_i), cross_{a_h,a_j}(V_h,V_j),$$
$$\quad dom_{a_h}(V_h), dom_{a_j}(V_j), dom_{a_j}(V_j).$$

For instance, we include the rule

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
name_visits(Name,Visits) :-
    name(Name), visits(Visits), position(Position),
    name_position(Name,Position) ,
    position_visits(Position,Visits) .
```

This rule says that if a person (like Robert) is in some position (like 2), and that position is associated with some planned destination to visit (like Martinique), then that person plans to visit that destination.

Partitions require a more permissive version of the transitivity property. For every two classes a_i, a_i, and every partition attribute p, we include in $Tr(P)$ only two rules:

$$cross_{a_i,p}(V_i, V_p) \coloneq cross_{a_i,a_j}(V_i, V_j), cross_{a_j,p}(V_j, V_p),$$
$$dom_{a_i}(V_i), dom_{a_j}(V_j), dom_p(V_p).$$
$$cross_{a_j,p}(V_j, V_p) \coloneq cross_{a_i,a_j}(V_i, V_j), cross_{a_i,p}(V_i, V_p),$$
$$dom_{a_i}(V_i), dom_{a_j}(V_j), dom_p(V_p).$$

Given these definitions and constraints, the attribute and cross-class predicates appearing in a stable model of $Tr(P)$ uniquely determine a table that satisfies the requirements of the declarations given by the Constraint Lingo program P. Conversely, each such table determines a stable model of the program $Tr(P)$.

The remaining part of the Constraint Lingo program consists of constraints specified by keywords such as `REQUIRED` and `CONFLICT`. To continue the description of the translation, we specify how these individual constraints are represented in the syntax of *smodels*.

(8) `CONFLICT` m_a m_b, where m_a m_b are elements of the domains of classes a and b, respectively. The role of this constraint in Constraint Lingo is to eliminate tables that contain rows with elements m_a and m_b in their corresponding columns. For each such constraint, we add the following rule to $Tr(P)$.

$$\coloneq cross_{a,b}(m_a, m_b).$$

In our case, we have

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

This rule means that no solution (the left-hand side is empty) may place `1` and `quebec` in the same row.

We extend this translation when the list of conflicting elements is longer than 2 elements; each pair of elements on the list gives rise to a constraint on the relevant cross-class predicate.

(9) `REQUIRED` m_a m_b. For each such constraint, we add the following rule to $Tr(P)$.

$$cross_{a,b}(m_a, m_b).$$

For instance, `REQUIRED quebec blueberry` would be translated as:

```
soda_visits(blueberry,quebec) .
```

Copyright © maybe later John Wiley & Sons, Ltd.

This fact indicates that any solution must place `blueberry` and `quebec` in the same row.

Again, we extend this translation when more than two members are listed.

(10) `VAR x`. One list class, say a, is selected arbitrarily. The variable `x` is meant to represent exactly one (unspecified as yet) element of that class. We introduce a new predicate $variable_x$ that holds just for that one element and build a rule that enforces that constraint:

$$1\{variable_x(X) : dom_a(X)\}1.$$

We represent the `unlemon` variable by using `name` as the arbitrarily chosen class and translating to:

```
1 {variable_unlemon(X):name(X)} 1 .
```

(11) `USED` n `<=` $partitionelement$ `<=` m. One list class, say a, is selected arbitrarily. There must be between n and m elements m_a in the domain of class a for which $cross_{a,p}(m_a, partitionelement)$ holds, where p is the partition to which $partitionelement$ belongs. We build the following rule to enforce this constraint:

$$n\{cross_{a,p}(A, partitionelement) : dom_a(A)\}m.$$

For instance, we would translate `USED 2 <= women <= 3` as:

```
2 {gender_visits(women,Visits) : visits(Visits)} 3 .
```

(12) Similar translations are easy to design for all the remaining constructs of Constraint Lingo. For the sake of brevity, we do not discuss them here. The interested reader may acquire our compiler [17] and inspect its output.

We believe our translation is correct: Let P be a Constraint Lingo program. For every table $T \in Sol(P)$, there is a stable model M of $Tr(P)$ such that M represents T. Conversely, for every stable model of $Tr(P)$ there is a table $T \in Sol(P)$ such that M represents T.

6. Optimizing the *smodels* translation

As compiler writers have known for years, there are many different correct translations for a given code fragment. High-quality compilers attempt to generate code that is especially efficient (in space and/or in time). Code optimization is also possible for Constraint Lingo. As we developed our compiler, we tried various alternative translations, settling on ones that give the fastest execution under *smodels*. In addition to minor adjustments to the translated code, we have also experimented with two fundamentally different approaches to $Tr(P)$.

We call the first new approach the **prime-class** representation. We arbitrarily choose the first list class as “prime”. We generate cross-class predicates in $Tr(P)$ only for pairs one of whose members is the prime class. We no longer need rules for transitivity, reducing the number of rules in the theory. However, constraints between elements of non-prime (“oblique”) attributes generate more complex rules, because they must be related via the prime class.

In the French puzzle, if `name` is prime, we translate `CONFLICT quebec 1` to

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
:- position_name(1, N), visits_name(quebec, N), name(N) .
```

In other words, instead of using the atom `position_visits(1,quebec)` (which is no longer available) we represent this constraint by joining `position_name(1, N)` and `visits_name(quebec, N)`. We can directly specify constraints involving members of the prime class and members of oblique classes. Our compiler uses a 3×3 case statement to cover all cases where the two members participating in a constraint belong to the prime class, an oblique class, or are variables.

Instead of choosing the prime class arbitrarily, we have implemented a variant called the **special-handle** translation in which the prime class is chosen after a first pass through the Constraint Lingo program to derive a weighted value for each class based on how often it is referenced and in what ways. This translation often generates the fastest code. We have tried other representations as well, but they don't behave as well as the ones we have introduced.

We present some comparisons of these optimizations with our original code in Section 8.

7. Other logic engines

We have described the *smodels* translation in some detail. Constraint Lingo is not specific, however, to *smodels*; we also have translators that convert programs into other logic formalisms. Each logic formalism requires that the translator writer study its syntax and semantics in order to generate a quality translation. This effort is often quite extensive. We claim that the person trying to solve a tCSP should not be required to expend this effort; it is all done once and is embedded in the translators.

We now touch on two of the logic engines beside *smodels* that we have used: disjunctive-logic programming and constraint-logic programming. In interests of space, we do not discuss a third logic engine: the logic of propositional schemata and its solver *aspps* [10].

7.1. Translation for disjunctive logic programming

The *dlv* logic engine [18] accepts much the same syntax as *smodels*, so our translation into *dlv* looks similar for most of Constraint Lingo. However, *dlv* does not have cardinality constraints, so the rules that guarantee uniqueness of cross-class predicate solutions are more complex than the one shown in Section 5 for *smodels*. For instance, we would translate `USED 3 <= men` in the French puzzle as:

```
counter(0) .
counter(1) .
counter(2) .
counter(3) .
counter(4) .
counter(5) .
atleastmen(none, 0) .
atleastmen(claude, N) :- atleastmen(none, N), N < 1, counter(N) .
atleastmen(claude, M) :- atleastmen(none, N), M = N+1,
    gender_person(men,claude), N < 1, counter(N) .
```

Copyright © maybe later John Wiley & Sons, Ltd.

```
atleastmen(jeanne, N) :- atleastmen(claude, N), N < 2, counter(N) .
atleastmen(jeanne, M) :- atleastmen(claude, N), M = N+1,
    gender_person(men,jeanne), N < 2, counter(N) .
atleastmen(kate, N) :- atleastmen(jeanne, N), N < 3, counter(N) .
atleastmen(kate, M) :- atleastmen(jeanne, N), M = N+1,
    gender_person(men,kate), N < 3, counter(N) .
atleastmen(liana, N) :- atleastmen(kate, N), N < 4, counter(N) .
atleastmen(liana, M) :- atleastmen(kate, N), M = N+1,
    gender_person(men,liana), N < 4, counter(N) .
atleastmen(martin, N) :- atleastmen(liana, N), N < 5, counter(N) .
atleastmen(martin, M) :- atleastmen(liana, N), M = N+1,
    gender_person(men,martin), N < 5, counter(N) .
atleastmen(robert, N) :- atleastmen(martin, N), N < 6, counter(N) .
atleastmen(robert, M) :- atleastmen(martin, N), M = N+1,
    gender_person(men,robert), N < 6, counter(N) .
:- not atleastmen(robert, 3) .
```

We have sorted the people; Robert turns out to be the last one. So the predicate `atleastmen(robert,N)` indicates that at least N of the people were men. The last rule then constrains this count.

Existence is assured by disjunctive rules, such as

```
position_visits(1, quebec) v position_visits(1, tahiti) v
     position_visits(1, haiti) v position_visits(1, martinique) v
     position_visits(1, belgium) v position_visits(1, ivory).
```

Disjunctions also assist in generating good code for the `MATCH` constraint.

The prime-class and special-handle compile-time optimizations of Section 6 also apply to disjunctive logic programming. Further details of the translation can be found in our compiler [17].

7.2. Translation for constraint-logic programming

Our approach to solving tabular CSP problems is different from the classical approach in the logic community, which is to directly represent such problems as constraints in constraint-programming languages. The logic puzzles solved by Doug Edmunds [19], for example, are all hand-coded. Our experience, however, is that it is far easier to program such problems in Constraint Lingo and then translate them into whatever form is appropriate for the computational engine. In keeping with that approach, we have built a translator from Constraint Lingo to ECL^{i}PS^{e}. Complete details can be found in our compiler [17].

The resulting ECL^{i}PS^{e} program is a single rule with many clauses on its right-hand side. We represent each row of the result table by an integer index ranging from 1 to the number of rows. In the French puzzle, the classes `name`, `visits`, and `position` are represented as multiple clauses of a single rule, as follows:

```
Name = [Claude, Jeanne, Kate, Liana, Martin, Robert],
Name :: 1..6,
alldifferent(Name),
Visits = [Quebec, Tahiti, Haiti, Martinique, Belgium, Ivory],
```

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
Visits :: 1..6,
alldifferent(Visits),
Position = [Position1, Position2, Position3, Position4, Position5,
    Position6],
Position :: 1..6,
alldifferent(Position),
```

If there is no numeric class, we break symmetry by selecting one class (such as `name`) as prime and assigning each member to a particular row:

```
Claude = 1, Jeanne = 2, Kate = 3, Liana = 4, Martin = 5, Robert = 6,
```

If one is available, we select a numeric class as prime and use its elements as row numbers. (A numeric class is only available if all its elements are used.) In the case of the French puzzle, `position`, which is numeric, is a better prime class than `name`, which is not. Order constraints involving a numeric class are much more efficient to represent if that class is prime.

Complex Constraint Lingo constraints such as

```
REQUIRED quebec blueberry OR quebec lemon
```

are represented simply as

```
Quebec #= Blueberry #\/ Quebec #= Lemon
```

Because `position` is the prime class,

```
CONFLICT quebec 1
```

is represented as

```
1 #\= Quebec
```

If we select `name` as the prime class instead, then this constraint becomes

```
Position1 #\= Quebec
```

Ordering relations involving a numeric prime class are quite easy. For instance,

```
OFFSET !+-1 position: robert kate
```

becomes

```
Robert + 1 #\= Kate #/\ Robert - 1 #\= Kate #/\
Robert + 1 - 6 #\= Kate #/\ Robert - 1 + 6 #\= Kate
```

If the ordering relation is with respect to an oblique class, the code is clumsier and lengthier, including parts like this:

```
(Robert #= Position1 #/\ Kate #= Position1) #\/
(Robert #= Position1 #/\ Kate #= Position3) #\/
(Robert #= Position1 #/\ Kate #= Position4) #\/
(Robert #= Position1 #/\ Kate #= Position5) #\/
(Robert #= Position2 #/\ Kate #= Position2) #\/
(Robert #= Position2 #/\ Kate #= Position4) #\/
(Robert #= Position2 #/\ Kate #= Position5) #\/
(Robert #= Position2 #/\ Kate #= Position6) #\/ ...
```

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

Partitions are clumsy to represent. For gender, we introduce the following:

```
Gender = [Gender1, Gender2, Gender3, Gender4, Gender5, Gender6],
Gender :: ['Men', 'Women']
```

Then we translate constraints such as

```
AGREE men: claude
```

into

```
(Claude #= 1 #/\ Gender1 #= 'Men' #\/
Claude #= 2 #/\ Gender2 #= 'Men' #\/
Claude #= 3 #/\ Gender3 #= 'Men' #\/
Claude #= 4 #/\ Gender4 #= 'Men' #\/
Claude #= 5 #/\ Gender5 #= 'Men' #\/
Claude #= 6 #/\ Gender6 #= 'Men')
```

The efficiency of ECL^{i}PS^{e} is quite sensitive to the heuristics explicitly indicated in the translated program; we have found that the best all-around choice is to use the *fd_global* library and to specify the "occurrence/indomain/complete" heuristic combination. It is likely that hand-tuning the programs would make them faster.

8. Efficiency tests

We have experimented with the following logic engines and representations: *smodels*[§] (cross-class, prime-class, special-handle), *dlv*[¶] (cross-class, special-handle), ECL^{i}PS^{e}[‖], and *aspps* (cross-class, special-handle). Our tests include (1) about 150 puzzles from Dell Logic Puzzles and Randall L. Whipkey [20] encoded in Constraint Lingo, (2) the independent-set graph problem on random graphs with 52 vertices and 100 edges, looking for 25 independent vertices, and (3) the 3-coloring problem on large random graphs.

All our tests ignore the time to compile Constraint Lingo programs (the compiler takes negligible time) and the grounding time for the logic engine (usually also negligible).

Our first conclusion is that the special-handle translation is usually far better than the cross-class translation. The following shows a few extreme examples of this trend; times are in seconds:

puzzle	logic engine	cross-class	special-handle
comedian	*aspps*	33	0.1
foodcourt	*dlv*	117	0.4
employee	*smodels*	38	0.4

§ *lparse* version 1.0.11; *smodels* version 2.27

¶ version BEN/Apr 18 2002. This version of *dlv* does not include cardinality constraints, unlike *smodels* and *aspps*.

‖ version 5.4, build 41

Copyright © maybe later John Wiley & Sons, Ltd.

Choosing the right translation is a matter of optimization. Even an expert logic programmer might create cross-class programs, because they often lead to shorter rules. Automatically performing this optimization leads to far more efficient code. We continue to find new optimizations.

Our second conclusion is that no one logic engine consistently outperforms the others, although *aspps* tends to be slightly faster than the others, and ECL^{i}PS^{e} tends to be slightly slower, failing to finish in a reasonable amount of time on a few puzzles. The following table compares the logic engines on our hardest puzzles; in all cases we show times for the special-handle translation, except for ECL^{i}PS^{e}, where the translation is completely different. We mark the "winner" in each case with a box; differences in time less than 0.05 seconds are most likely insignificant. We make no claim that our translations are optimal. These tests are not meant to demonstrate superiority of one logic engine over another, only to show the feasibility of our approach.

puzzle	*aspps*	*smodels*	*dlv*	ECL^{i}PS^{e}
card	[0.00]	0.01	0.01	0.02
comedian	[0.05]	0.12	2.29	0.78
employee	[0.24]	0.44	3.12	—
flight	0.01	[0.00]	0.01	0.04
foodcourt	[0.17]	0.54	0.41	3.13
french	[0.00]	0.03	0.08	0.13
jazz	[0.00]	0.04	0.03	0.02
molly	[0.03]	0.04	0.15	0.33
post	[0.00]	0.02	0.04	0.51
ridge	11.56	8.14	[0.76]	—
sevendates	[0.03]	0.05	0.05	0.04

The independent-set problem is represented, as shown in Section 4, by a `REQUIRED` constraint for each edge and a single `USED` constraint. Both *aspps* and *smodels* provide a notation that allows us to translate `USED` in P into a single cardinality constraint in $Tr(P)$. These logic engines enforce cardinality constraints during the search process, which leads to very efficient search. In contrast, neither *dlv* nor ECL^{i}PS^{e} provides cardinality constraints, so we program `USED` by explicitly counting how many times the desired member is used and then constraining that total. We can count directly in ECL^{i}PS^{e} and indirectly by extra rules in *dlv*. In both cases, this generate-and-check strategy (as opposed to a built-in construct) leads to slower searches.

The following table shows the number of seconds for several logic engines to compute the first model of a theory representing the independent-set problem looking for I independent vertices on a random graph with a V vertices and E edges. Again, we ignore compilation and grounding time.

V	**E**	**I**	*aspps*	*smodels*	*dlv*	ECL^{i}PS^{e}
100	200	40	0.01	1.08	1.61	60
100	200	44	18.99	25.19	148.6	394

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls

We continue to search for translations that perform better than our current ones. Our experience reinforces our belief that efficient solution of constraint-satisfaction problems depends on a carefully designed compilation; even experienced logic programmers are unlikely to achieve efficient programs without enormous effort.

9. Discussion and conclusions

Logic programming was introduced with a promise of dramatically changing the way we program. Logic programming is declarative. The programmer can solve a problem by encoding its specifications in some logic formalism and then invoking automated reasoning techniques for that logic to produce a solution. Control details are no longer the programmer's responsibility.

However, despite attractive features stemming from its declarative nature, logic programming has not yet gained a widespread acceptance in the programming world. This disappointing result seems to hold both for logic-programming implementations based on proof-finding techniques (*Prolog* and its extensions that handle constraint programming, such as ECL^{i}PS^{e}) and to newly emerging approaches based on satisfiability testing and model computation (answer-set programming [6, 5]).

This state of affairs is due to the fact that logic-programming formalisms are too low-level to be used without great effort and require that programmers have a significant logic background. In order to be successful, a declarative programming language should be aligned with language constructs often used when problems are described in free text. We suggest that programs in such a high-level language should be automatically compiled to programs in low-level languages such as current implementations of logic programming (*Prolog*, ECL^{i}PS^{e}, *smodels*, and so forth) and then solved by the corresponding solvers.

Our main contribution is a high-level declarative language, Constraint Lingo, designed to capture tabular constraint-satisfaction problems. Constraint Lingo is simple. It uses two constructs, `CLASS` and `PARTITION`, to define the framework in which a given problem is described, and 10 constructs to describe constraints, all of them well attuned to free-text descriptions of constraint problems.

We don't claim that Constraint Lingo is the best possible language for this purpose. Its line-oriented commands, each starting with a capitalized keyword, may appear a throwback to languages like Basic. Constraint Lingo has a limited repertoire of connectives and arithmetic operations; it has no general-purpose arithmetic or Boolean expressions.

Despite these limitations, Constraint Lingo is an expressive language in which one can describe a diverse collection of tabular constraint-satisfaction problems. We have used it to represent over 150 logic puzzles ranging in difficulty from one to five stars and involving a large variety of constraints, as well as several graph problems over randomly-generated graphs of various sizes. Thanks to its simplicity and affinity to free-text constraint specifications, programming in Constraint Lingo is easy and frees the programmer from many tedious and error-prone tasks.

Constraint Lingo provides a computational as well as a descriptive facility. We compile Constraint Lingo programs into executable code in a variety of low-level logic programming languages.

Copyright © maybe later John Wiley & Sons, Ltd.

Evidence shows that our approach is practical. Programs we obtain by automatically compiling Constraint Lingo programs closely resemble those that programmers have written directly. Our computational results are encouraging and show that programs produced by compiling Constraint Lingo programs perform well when processed by various computational engines.

We continue to evolve Constraint Lingo and its associated tools. Recent developments include the following, all available in the most recent release of the software [17].

- **Other constraints.** New syntax allows mappings between rows; these mappings can be declared to be nonreflexive, symmetric, asymmetric, and/or onto. This facility lets us represent some complex constraints, such as "Everyone has a hero in the room; Jeanne's hero is Kate, but Kate's hero is not Jeanne." Two maps can be declared to differ on every row, so we can indicate constraints such as "Nobody's hero is his or her tennis partner." We also have introduced syntax to indicate that the values of two partitions taken together act as a key, so we can indicate constraints such as "although a every floor has several rooms and every wing has several rooms, each room has a unique combination of floor and wing."
- **Problem-construction tools.** We have built a Tcl/Tk [21] front end to our Constraint Lingo package that allows us to build problems by (1) constructing the desired solution, (2) introducing constraints, (3) ensuring that the constraints so far are not contradictory (leading to no solutions) and are consistent with the desired solution, (4) identifying undesired solution components, (5) identifying superfluous constraints. We have used these tools to build extremely difficult puzzles, perhaps beyond human ability to solve.
- **Explanations.** We have instrumented the grounder and solver of *aspps* to generate a log file that we then convert into a set of steps that a human can follow to solve the problem. We have introduced new Constraint Lingo syntax that allows the programmer to specify how cross-class predicates are to be expressed in English. An explanation of the French puzzle includes deriving a conflict between `martinique` and `6`, and the English expression is, "to be consistent [explaining how this result follows from the previous results], the person sitting in seat 6 doesn't plan to visit Martinique [an English clause]."

We are considering other enhancements as well. The current support for variables is limited and might be extended to support universal quantification. Our implementation does not support data-input operations. It provides only restricted support for logical and arithmetic operations. We need better support for arithmetic if Constraint Lingo is to be applicable in modeling and solving real-life operations-research problems. However, as we contemplate extensions to Constraint Lingo, we want to be careful to preserve its simplicity, which, we believe, is its main strength.

A similar approach, finding helpful higher-level abstractions, might well be helpful in other structured domains, such as planning and scheduling. We have begun to look at both.

ACKNOWLEDGEMENT

We thank Hemantha Ponnuru and Vijay Chintalapati for their programming and testing support.

Copyright © maybe later John Wiley & Sons, Ltd.

REFERENCES

1. R. Kowalski. Predicate logic as a programming language. In *Proceedings of the Congress of the International Federation for Information Processing (IFIP-1974)*, pages 569–574, Amsterdam, 1974. North Holland.
2. A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un systeme de communication homme-machine en francais. Technical report, University of Marseille, 1973.
3. K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, MA, 1998.
4. M. Wallace, S. Novello, and J. Schimpf. Ecl$^{\rm i}$ps^e: A platform for constraint logic programming, 1997. http://www.icparc.ic.ac.uk/eclipse/reports/eclipse.ps.gz.
5. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
6. V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, Berlin, 1999.
7. I. Niemelä and P. Simons. Extending the smodels system with cardinality and weight constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.
8. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
9. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the 6th International Conference on Knowledge Representation and Reasoning (KR-1998)*, pages 406–417. Morgan Kaufmann, 1998.
10. D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 138–153. Springer, 2001.
11. Christiane Bracchi, Christophe Gefflot, and Frederic Paulin. Combining propagation information and search tree visualization using ILOG OPL studio. November 16 2001. In A. Kusalik (ed), Proceedings of the Eleventh International Workshop on Logic Programming Environments (WLPE'01), December 1, 2001, Paphos, Cyprus, cs.PL/0111042.
12. B. Selman and H. A. Kautz. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, Vienna, Austria, 1992.
13. H.A. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proceedings of IJCAI-99*, San Mateo, CA, 1999. Morgan Kaufmann.
14. R.E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal*, 2:189–208, 1971.
15. N.J. Nilsson. *Artificial Intelligence: a New Synthesis*. Morgan Kaufmann, San Francisco, 1998.
16. M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl – the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, October 1998. Available at http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz.
17. R. Finkel. Constraint lingo package, 2003. ftp://ftp.cs.uky.edu/cs/software/cl.tar.gz.
18. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving in DLV. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, Dordrecht, 2000.
19. D. Edmunds. Learning constraint logic programming — Finite domains with logic puzzles, 2000. http://brownbuffalo.sourceforge.net.
20. R. L. Whipkey. Various logic puzzles, 2001. http://www.allstarpuzzles.com/logic and http://crpuzzles.com/logic.
21. John K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994. ISBN 0-201-63337-X.

Copyright © maybe later John Wiley & Sons, Ltd.

Prepared using speauth.cls