A Decade of Progress in Constraint Modelling and Reformulation
The Quest for Abstraction and Automation

Alan M Frisch
Artificial Intelligence Group
Dept of Computer Science
University of York
Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, and the computer solves it.  

[Freuder, Constraints ’97]

It is one thing to say ‘all one has to do is express the problem constraints.’ It is another to express them in a manner which permits efficient solution”  

[Freuder, Constraints ’97]
Achieving the Holy Grail

It is one thing to say ‘all one has to do is express the problem constraints.’ It is another to express them in a manner which permits efficient solution.”

In fact ... It can be difficult to express the problem constraints in a manner which permits even inefficient solution.

“...and that, in simple terms, is my idea on how to increase factory optimization. any questions?”
Achieving the Holy Grail

Problem

Modelling

Constraint Solver

Solutions

Abstraction

Automation
Abstraction
Declarative Languages

• A big story over last decade: the rise of declarative languages for specifying models and problems.

• Among the declarative languages, Constraint-based languages (DCLs) have gained a dominant position
  - Constraints over decision variables with domains
  - Ever richer collection of domains and constraints
  - SAT and ILP are “degenerate” cases

• Not just for use with FD-constraint solvers: Also SAT, SMT, ILP, hybrid solvers (SCIP), local search solvers (COMET)

• There are alternatives: NP-Spec, Alloy, Z, Depict, MX
Declarative Constraint Language (DCL)

- Four main components of a specification in such a language

1. **Parameters** - what comprises an instance
   
   e.g.: given N: int+, V: array[1..N] of int

2. **Decision variables and their domain**
   
   e.g.: Find X: 1..10, A: array[1..N] of 0..2*N-1

3. **Constraints on the variables**
   
   e.g.: for all i in 1..N-1. A[i] < A[i+1]

4. **Objective function** - if an optimisation problem
   
   e.g.: Maximise Sum i:1..N. A[i]
Why Declarative Constraint Languages Dominate

• This is how people (at least computer scientists) think of problems.

• Evidence: The 300+ specifications of NP-complete problems in Garey and Johnson [A Guide to the Theory of NP-Completeness, ’79]

INSTANCE: Finite set U, for each \( u \in U \): a size \( s(u) \in \mathbb{Z}^+ \), a value \( v(u) \in \mathbb{Z}^+ \) and positive integers \( B \) and \( K \).

QUESTION: Is there a subset \( U' \subseteq U \) such that \( \sum_{u \in U'} s(u) \leq B \), and \( \sum_{u \in U'} v(u) \geq K \)?

• Further evidence: the 70+ problem specifications written in ESSENCE (a declarative constraint language) by two undergrads with no knowledge of CP [http://www.cs.york.ac.uk/aig/constraints/AutoModel/Essence/specs120/]
History of Development of Declarative Constraint Languages: Direct Systems

- Inherited from GAMS and MGG mathematical programming languages. Later AMPL.

- Specify parameterised models of *problems* and (separately) instance data.

- Schemas showing the pattern of the constraint set as a function of the parameter values.

- Key ideas illustrated by considering how to construct a direct DCL for a solver.
### How to Build a Direct Language for a Given Solver

#### 1. Base Case:
- Put all solver-supported constraints in the language.
  \[ X + Y + Z = 12 \]

#### 2. Parameterise:
- Add parameters, enabling problem classes rather than instances to be specified.
  \[ \text{param } t; \quad X + Y + Z = t \]
- Allow values for the parameters to be specified separately. This is used to specify instances.
  \[ t = 12 \]

#### 3. Lift:
- Add arrays (indexed variables) enabling different instances to have different numbers of variables.
  \[ \text{param } c; \quad \text{var } X[1..c]: \text{int} \]
- Add schemas enabling different instances to have expressions of different sizes.
  \[ \sum_{i=1..c} X[i] = t \]
How to Compile a Direct Language

```
param t;  param c;
var X[1..c]: int(1..t)
∑_{i=1}^{c} X[i] = t
```

```
t=12;  c=3;
```

```
var X[1..3]: int(1..12)
∑_{i=1}^{3} X[i] = 12
```

```
var X1, X2, X3: int(1..12)
X1 + X2 + X3 = 12
```

- Instantiate parameters
- Unroll arrays and schemas
OPL: State-of-the-art DCL at Time Zero (a decade ago)

[Van Hentenryck, The OPL Optimization Programming Language, ’99]

- Mostly used direct approach, but with some extensions

- Three solving algorithms:
  - MIP: for pure MIP problems
  - Solver: for FD-constraint problems, including linear constraints
  - Solver/LP: Solver using linear relaxation of MIP constraints in two hybrid schemes: “bound cooperation” and “cut cooperation”

- No “cross compiling”: only linear constraints used in MIP or LP
  - No linear representations of non-linear/global constraints
  - But user can supply them by building a redundant model w/ channeling
    [Van Hentenryck, INFORMS JoC ’99]
  - Can combine different models for different parts of a problem
    [Van Hentenryck, Michel, Perron, Regin, PDDP’99]

- Provides scheduling primitives (naturalness through abstraction)
Past Decade of Research: DCLs Beyond the Direct Approach

Advances in three areas:

(1) Decouple DCLs from FD-solvers
   - generate input for other kinds of solvers
   - one DCL to generate input for multiple solvers

(2) More domains, especially
   - domains not supported by solvers
   - non-atomic domains

(3) Quantification ≠ Schema for a set of expressions

• Taken far enough, the result is a problem specification language
(1) Decoupling DCLs from Solvers

• Minizinc
  - G12 FD constraint solver [Becket et al, ModRef’08]
  - Yices SMT solver: Fzn2smt system [Bofill, Soy, Villaret, SAT’10]
  - SAT: Fzntini system [Huang, CP’08]
  - Chuffed lazy clause solver [Ohrimenko, Stuckey, Codish, Constraints ’09]
  - SCIP hybrid CP/MIP solver [Achterberg, MPC1’09]
  - Cplex MIP solver [Brand, Duck, Puchinger, Stuckey, PADL’08]
  - constraint programming languages: Gecode, ECLiPSe, SICStus Prolog, JaCop
(1) Decoupling DCLs from Solvers

• COMET $\Rightarrow$
  - constraint-based local search solver
  - constructive-search backtrack solver

• XCSP $\Rightarrow$ SAT w/ order encoding: Sugar system
  [Tamura, Taga Kitagawa, Banbara, Constraints, '08]

• and more.....
Decoupling DCLs from Solvers: Pipelining!
(1) Decoupling DCLs from Solvers: Pipelining!
From its original conception, CONJURE was designed to map ESSENCE to ESSENCE' and then ESSENCE' to multiple solvers.
(1) Decoupling DCLs from Solvers: Pipelining!

Popularised and hugely exploited by MiniZinc
Tailor supports multiple inputs and multiple outputs.  [Rendl, PhD Dissertation ’10]
(2) More Domains: Non-atomic Domains

- ECLiPSe: sets

- $\mathcal{F}$: sets, functions (partial and total), total functions, permutations, sequences, and bounded sequences

- ESRA: functions, relations [Flener, Pearson, Agren, ModRef’03]

- NP-Spec: sets, permutations, partitions, functions

- MiniZinc: sets

- Zinc constructors: set, record, tuple, discriminated union, array

- ESSENCE constructors: set, multiset, array, partition, tuple relation, function
  - and many specialisations of these (e.g., total function, regular partition)
(2) More Domains: Atomic Domains

- Traditional atomic domains: Integers, Booleans, enumerated types
- New: unnamed elements [Frisch, Hernandez, Jefferson, Miguel, IJCAI'07]
- Example: Social golfers Problem: Partition $g$ golfers into ......
  - Model 1: Let golfers be 1..$g$
  - Model 2: Let golfers be new type (size $g$)
(3) Quantification ≠ Schema for a Set of Expressions

- Quantification over decision variables available in ESRA, \( F \), Localizer and ESSENCE.

- Example:

Find a 5 element set \( S \) of \( 1..10^6 \) such that \( \sum_{e \in S} e^2 = 10^{12} \)
Illustration of importance:

**With**

Find a 5 element set $S$ of $1..10^6$
Such that $\sum_{e \in S} e^2 = 10^{12}$

Find $A$ such that $\text{alldifferent}[A]$
$\sum_{i \in 1..5} A[i]^2 = 10^{12}$

**Without**

Find a 5 element set $S$ of $1..10^6$
Such that $\sum_{i \in 1..10^6} (\text{bool2int}(i \in S) \times i^2) = 10^{12}$

unrolls into sum of $10^6$ terms

unrolls into sum of 5 terms
Final Note on Abstraction:
An Indication That a DCL Lacks Some Facility for Abstraction

If a problem can’t be modelled in a DCL without *introducing* additional symmetries, then the DCL is lacking some kind of abstraction.

Examples:

• If DCL doesn’t have domains of unnamed elements then a named domain must be used. These named values can be swapped.

• If DCL doesn’t have domains of type “set of $\tau$” and “list of $\tau$” used instead then elements in the list can be swapped.
Automation
The Modelling Process

1. Encode the problem constraints, including variables/domains (kernel)
   - may involve redundant representations and channelling constraints

2. Add symmetry-breaking constraints and constraints that follow from dominance

3. Add implied constraints
   - often implied in part by symmetry-breaking constraints
     [Frisch, Jefferson, Miguel, ECAI'04]

4. Perform efficiency-enhancing transformations
Automation in: Modelling: Advances in Five Areas

- Generating a kernel
  - Compilation in a direct system
  - Compilation in a non-direct system
  - Machine learning: active research area, not discussed here

- Identifying symmetries
- Breaking symmetries
- Adding implied constraints
- Transforming constraints
Generating a Kernel: Compilation in a Direct System

• Limited abstraction (and unique representations) gives user control over the model produced

• Still the compiler can
  - perform efficiency-enhancing transformations
  - add symmetry-breaking and implied constraints (assuming the model has not already broken symmetry)

• State of the art
  - All compilers (as far as I know) translate problem instances, not problem classes.
  - A great deal of implementation work, yet little published.
    - notable exception is Tailor [Rendl, PhD’09]
  - Open questions remain:
    - E.g.: should a prenex $\exists$ be implemented as a disjunction or a decision variable? Or should user decide? [Jefferson, Petrie, ModRef’08]
Generating a Kernel: Compilation in a Non-Direct System

• Necessary for: ESRA, $\mathcal{F}$, Zinc, ESSENCE
  - Notable that all these have compilers that handle problem classes

• All the issues of compilation in a direct system, plus the
  - need to encode those abstract constructions not supported by solvers
  - All details of the generated model are not visible in the input specification so in many cases the compiler **must** be the one to
    - infer and add implied and symmetry-breaking constraints
    - perform efficiency-enhancing transformations
Generating a Kernel: Compilation in a Non-Direct System

- The common, straightforward approach for “compilers” is to generate a fixed representation for each kind of variable not supported by the solver.
  - Taken farthest by Reza Rafeh’s [PhD'08] system: generates occurrence representation for all non-atomic variables in Zinc.

- Both $F$ and CONJURE generate multiple representations for non-atomic variables.
  - $F$ can use “canned” representations, but CONJURE must construct them since it has an infinite set of variable types.

- Zinc [De Koninck, Brand, Stuckey, ModRef'10] generates only one representation
  - Like CONJURE, uses rules (though very different rules)
  - Future: able to generate multiple representations, user provides annotations to determine selection
Generating a Kernel: Redundant Models and Channelling

- The $F$ compiler, Fiona, includes redundant models among the canned representations it has for decision variables.

- The rules of CONJURE can generate redundant models that include any number of representations of a decision variable. CONJURE also generates the appropriate channelling constraints.

[Martinez-Hernandez, PhD]
Identifying Symmetries in Problem Instances

- Constraint symmetries in a constraint model polynomially reduces to automorphisms in a graph
  - Introduced by Crawford [Wkshp on Tractable Reasoning ’92]
  - Reduction improved and extended by others [Ramani, Markov, CSCLP’04] [Aloul, Markov, Sakallah, ASP DAC’04] [Puget, CP’05] [Mears, Garcia de la Banda, Wallace, Constraints’09]
  - Automorphism groups in **very large graphs** can be found automatically:
    - **nauty** [Congressus Numerantium ’81]
    - **Saucy2** [Darga, Sakallah, DAC’08]
    - **AUTOM** [Puget, CP’05]

- **CGRASS** uses ad-hoc techniques
  - Particular note: normalisation makes some symmetries visible as some semantic symmetries become syntactic symmetries [Frisch, Miguel, Walsh, Recent Advances in Constraints ’02]
Identifying Symmetries in Problem Classes

• Using a logical representation Joslin and Roy [AAAI’97] discover symmetries by reduction to graph automorphism. Theorem identifies conditions under which these symmetries map to the propositional level.

• Identify symmetries introduced when compiler compiles out abstract types [Frisch, Jefferson, Martinez-Hernandez Miguel, Int Symmetry Conf ’07]
  - Example: compiling “set (size c) of T” to “array [1..c] of T” introduces all symmetries that permute the elements of the array.
  - Example: compiling an unnamed type of n elements to 1..n introduces all symmetries that permute the values 1..n.
  - In compiling a nested type, the rules can compose the symmetries generated by each.
  - Example: SGP: decision variable is of type multiset of partition of golfers. The rewrite rules can identify all the symmetries.
Breaking Symmetries in Problem Instances

Breaking All Symmetries

- Lex-leader: a general method for breaking all variable symmetries [Crawford, Ginsberg, Luks, Roy, KR’96]
  - For each symmetry impose a lexicographic-ordering constraint, rules out all but one of the symmetric solutions.
  - This is often impractical in practice: too many constraints.

- In fact, maintaining arc consistency on constraints that break all variable symmetries is in general NP-complete. [Walsh, SymCon’11]

- For value symmetries, the story is similar. [Walsh, CP’07]
• A common approach is to add only those lex-ordering constraints associated with the group generators obtained by nauty [Aloul, Ramani, Markov, Sakallah, DAC’02] [Aloul, Sakallah, Markov, IEEE ToC’03] [Katsirelos, Narodytska, Walsh, SymCon’09]

• If group has too many cycles this can be also be impractical, so use only the first few cycles [Ramani, Markov, CSCLP’04]

• Jefferson and Petrie [CP’11] propose an alternative way of reducing the set of constraints. Evidence is that this generates more constraints than using all generators (or using lex²) but, in exchange, it is more effective.
Breaking Symmetries in Problem Classes

- No known published work.
- But it should be possible to do it as part of CONJURE-style compilation.
Adding Implied Constraints to a Problem Instance

- **CGRASS** [Frisch, Miguel, Walsh, CSCLP’02] uses a set of ad-hoc rewrite rules to transform a problem instance by:
  - normalisation
  - adding implied constraints and symmetry breaking constraints
  - reformulating a constraints to make them more efficient

- **Conclusions**
  - number of rule applications often grows very rapidly with size of the instance, so it is better to work with the lifted problem specification than the instances.
  - complex patterns of constraints often need to be handled, so it is better to work with a highly-abstract statement of the problem
  - this motivated us to start work on designing ESSENCE
Adding Implied Constraints to a Problem Class

- Problem Classes [Colton, Miguel, CP’01] [Charnley, Colton, Miguel, ECAI’06]
  - start with a logical specification of a problem class
  - generate and solve a number of small instances
  - discover interesting general properties of all the solutions (HR)
  - attempt to prove that the general properties logically follow from problem specification (Otter). If so it is an implied constraint for the problem class.
Transforming Constraints

- CGRASS as mentioned previously.

- More thorough treatment of transformations by Tailor \cite{Rendl, PhD'10}, which performs highly-effective efficiency-enhancing transformations:
  - replaces common subexpression by a new variable
  - removes duplicate constraints
  - reformulation to increase number of applications of above
  - quantifier optimisations before unrolling (e.g., moving invariant expressions outside scope of quantifiers)

- Tailor implementation handles instances, but transformations for problem classes have been formulated.
Future Issues
Five Issues that Have Barely Been Addressed

(1) Automatically Making Modelling Choices

(2) From Problem Instances to Problem Classes

(3) Generating search strategies

(4) Automatic generation of constraints from dominance reasoning

(5) Difficulties in modelling unbounded and unknown sizes

and some suggestions on how to proceed
(1) Automatically Making Modelling Choices

Fiona and CONJURE generate multiple models for an input specification (F and ESSENCE). Soon Zinc. None select from among the alternatives.

No research on how to automatically make modelling choices

Two possible approaches:

Human Learning

Researchers study the issue and embed the resulting expertise in automated modelling systems.

Machine Learning

An automated modelling system generates and tests performance of alternative models. Over time learns to generate the better models.

Suggestion: use machine learning
(2) From Problem Instances to Problem Classes

- Vast majority of work on automation of modelling considers (unrolled) problem instances, not problem classes.

- But unrolling often increases size greatly, which often renders reasoning infeasible. [Frisch, Miguel, Walsh, CSCLP'02]

- We need to pay significantly more attention to reasoning about problem classes.

- But formulating reasoning processes on uninstantiated parameters and especially schemas is often hard -- very hard.

- Suggestion: many of the difficulties of reasoning with problem classes can be mitigated by reasoning with a highly abstract representation of the problem.
(3) Specifying or Generating Search Strategy

- Achieving the holy grail requires a search strategy for solving automatically-generated models with an FD-constraint solver

- Approaches
  - Develop and use a universal search strategy (as is common in SAT solvers)
  - Automatically generate a strategy for each model (as in [van Hentenryck, Michel, AAAI'07] [Elsayed, Michel, CP’11])

- Suggestion: Significant advantage can be gained by reasoning with a highly abstract specification when automatically generating a search strategy
(4) Automatic Generation of Constraints from Dominance Reasoning

- There has been no research on this topic

- Suggestion: Consider “Minimize $f(x)$” as “$f(x) < c$” and “Maximize $f(x) > c$” as “$f(x) > c$”
(5) Difficulties in Modelling Unknown and Unbounded Sizes

**Golomb Ruler Problem:** The problem asks for the smallest ruler but gives no upper bound. Must provide one in a CP model.

**Template Design Problem:** Problem allows different number of templates to be considered. Usual model fixes the number. Same issue arises in Cover Test Problem and Planning Problem.

**Cover Test Problem:** A straightforward model [Hnich, Prestwich, Selensky, Smith, *Constraints*’06] produces anomalous behaviour. If allowed to perform more tests than needed, then time-to-solution increases!
Thanks for your attention

“It may be a model, Captain, but it’s highly illogical.”