Towards Schema-Guided Compilation of Set Constraint Programs

Pierre Flener, Brahim Hnich, and Zeynep Kızıltan
Computer Science Division, Department of Information Science
Uppsala University, Box 311, S – 751 05 Uppsala, Sweden
{pierref,brahim,zeykiz}@csd.uu.se

1 Introduction

Optimisation problems — where appropriate values for the variables of the problem have to be found, subject to some constraints, such that some cost function on these variables takes an optimal value — are ubiquitous in industry. Examples are production planning subject to demand and resource availability so that profit is maximised, air traffic control subject to safety protocols so that flight times are minimised, transportation scheduling subject to initial and final location of the goods and the transportation resources so that delivery time and fuel expenses are minimised, etc. A particular case are *decision problems*, where there is no cost function that has to take an optimal value. Many of these problems can be expressed as constraint programs and then be solved using constraint solvers.

However, effective *constraint programming* (CP) [19] is very difficult, even for application domain experts, and hence time-consuming. Moreover, many of these problems are ill-behaved, in the sense that it can be shown that solving them requires an amount of time that is worse than polynomial in the size of the input data, hence making solving times prohibitively long.

To address the *programming time problem*, ever more expressive programming languages[1] are being developed, providing traditional algebraic notations (such as sums and products over indexed expressions) and useful datatypes (such as sets, multi-sets, sequences, and enumerations) to enable a more natural expression of the constraints, freeing the programmer thus more and more from traditional (and often low-level) computing obligations, such as the writing of iterative/recursive code or the encoding of concepts as numbers. These languages are also increasingly declarative.[2] However, there is very little work on methodologies on how to use such languages.

To address the *solving time problem*, a search procedure and redundant constraints can be added so as to accelerate the default solution enumeration. Such optional (but often necessary) practice is however a concession that fully declarative programming is still far away, and the question whether these non-declarative statements can be automatically added upon analysis of the declarative ones remains essentially open. Concerns about the solving time also require trade-offs about expressiveness: the programming language must after all be executable (though need not be computationally complete) and its programs should ideally execute quickly (and finitely). For instance, set constraint languages may well allow the formulation of constraints over sets, hence yielding enormous expressiveness, but, as of now, there are only quite restricted and slow (academic) prototype languages, such as CLPS [1], COJUNTO [11], and NP-SPEC [4], as well as the fast OZ [26], which seems to be the only one to allow the formulation of constraints on non-ground sets of unknown (but finite) size [20]. To our knowledge, no set constraint language currently allows the formulation of constraints on possibly infinite sets or on sets whose elements are drawn from an infinite domain. If expressive declarative languages cannot be compiled into acceptably fast code, then the advantage of decreased programming time is neutralised by the disadvantage of increased solving time.

We call *modelling* the usage of a CP language for expressing an optimisation/decision problem. This results in a *model*, which thus has a *constraint part* and an optional *search part*.

We here report on our initial results regarding the compilation of very expressive, purely declarative, typed, first-order set constraint logic programs into clp(FD) programs [6] (clp(FD) being a very fast constraint logic programming solver over finite domains).

This paper is organised as follows. In Section 2, we discuss our set constraint programming language as well as (syntactic forms of programs of) the target language clp(FD). Then, in Section 3, we introduce the semantic notion of program schema, which captures entire families of similar programs. We are then set to explain, in Section 4, our notion of schema-guided compilation. Finally, in Section 5, we conclude, compare with related work, and discuss our directions of future work.

Disclaimer, Conventions, and Notation

For simplicity, we here only consider decision problems. We also ask the reader to bear with us whenever there are (often deliberate) simplifications, theoretical imprecisions, or vague terminology (set between single quotes), as we wish to get some novel ideas across without getting stuck in theoretical details and notational clutter, all

[1]Programming language A is *more expressive* than programming language B if for a program in A of n words there is an equivalent program in B of more than n words, *equivalence* of two programs being achieved when they always compute the same results from the same data.

[2]A programming language is *declarative* if its programs only describe *what* the properties of solutions are, but without expressing *how* these solutions can be found.

of which rather belong to a thorough and much more voluminous study.

In formulas, variable symbols start with an uppercase letter, whereas (non-numeric) constant, function, relation, and type symbols start with a lowercase letter. They are all typeset in *italics*. Unquantified variables are assumed to be universally quantified over the entire formula in which they occur. Whenever types are not so important, we omit them. When we wish to group (n) terms $t_1, \ldots, t_n$ into a single term, called an (n-) tuple, we use angled brackets, yielding $\langle t_1, \ldots, t_n \rangle$.

2 Programs

All programs, whether of the input or target language, are here theories in some typed, first-order logic. The input language, let us call it $\mathcal{S}$ until we have decided on a suitable name for it, is partially introduced below. The target language here is clp(FD), but our compilation technique is independent of that choice. We consider open (or: parametric) programs, together with the corresponding notion of 'open equivalence' (or: parametric equivalence). We consider the Horn-clausal notation for (open) (constraint) logic programs to be syntactic sugar for their typed (open) completions [17]. Hence the following definitions:

Definition 2.1 A relation symbol r occurring in a theory T in a language $\mathcal{L}$ is *open* in T if it is neither 'defined' in T, nor a primitive symbol in $\mathcal{L}$.
A non-open symbol in T is a *closed* symbol in T.
A theory with at least one open symbol is called an *open* theory; otherwise it is a *closed* theory.

Definition 2.2 A *program* for a relation r is a possibly open theory that 'defines' r.
Program P, whether open or closed, is a *refinement* of open program T *under* extension θ if θ is a set of programs 'defining' some of the open symbols of T such that P and $T \cup \theta$ are 'open-equivalent.'

Finding extensions can be done through second-order matching, which is decidable but NP-complete in general, though linear for higher-order patterns [15], where all predicate variables (or open relation symbols, here) apply to distinct variables only, which is the case here.

Note that closed programs thus do not have refinements. We ask the reader to overlook the fact that in our examples we use a much more general definition of refinements. Indeed, refinements do not necessarily have the same 'defined' relation symbol as the open program, refinements do not necessarily 'declare' their formal parameters in the same order as the open program, refinements may have more or less formal parameters than the open program, and the formal parameters of refinements may be of 'sub-types' of the types in the open program. Most of the considerations so far are illustrated in the following examples.

Example 2.1 Let us first look at our input language $\mathcal{S}$. The chosen representation of sets in $\mathcal{S}$, called the *external representation*, is the classical one, with curly braces.

Programs in $\mathcal{S}$ are *iff-programs,* expressing that, under input condition i_r on input X, a program for relation r must succeed if and only if output condition o_r on X and output Y holds. Formally, this gives rise to the following open program for r:

$$\begin{array}{c} \forall X : term \, . \, \forall Y : term \, . \\ i_r(X) \rightarrow (r(X,Y) \leftrightarrow o_r(X,Y)) \end{array} \qquad (\textit{iff})$$

The only open symbols are relations i_r and o_r. We here only consider programs for *subset decision problems*, where a subset S of a given finite set T (an integer set here) has to be found, such that S satisfies an (open) constraint g and an arbitrary two different elements of S satisfy an (open) constraint p. (Other problem families have been and will be handled similarly, see Section 4.) Hence the following open program:

$$\begin{array}{c} \forall T : set(int) \, . \, \forall S : set(T) \, . \\ subset(T,S) \leftrightarrow g(S) \wedge \\ \forall I,J : S \, . \, I \neq J \rightarrow p(I,J) \end{array} \qquad (subset_{dec})$$

The only open symbols are relations g and p (assuming that $\neq$ is a primitive of $\mathcal{S}$, with the usual meaning). Note how the type of S 'depends' on T. The open program $subset_{dec}$ is a refinement of open program *iff* above. It has itself as refinements programs for many problems, such as finding a clique of a graph (see below), set covering, knapsack, etc. For instance, the closed program

$$\begin{array}{c} \forall \langle V,A \rangle : set(int) \times set(V \times V) \, . \, \forall C : set(V) \, . \\ clique(\langle V,A \rangle, C) \leftrightarrow sum(C,S) \wedge S > 22 \wedge \\ \forall M,N : C \, . \, M \neq N \rightarrow \langle M,N \rangle \in A \end{array} \qquad (clique_{dec})$$

is a refinement of $subset_{dec}$, under the extension

$$\begin{array}{l} \forall V : set(int) \, . \, \forall C : set(V) \, . \\ g(C) \leftrightarrow sum(C,S) \wedge S > 22 \\ \\ \forall \langle V,A \rangle : set(int) \times set(V \times V) \, . \, \forall C : set(V) \, . \\ \forall I,J : C \, . \, p(I,J) \leftrightarrow \langle I,J \rangle \in A \end{array} \qquad (\sigma)$$

assuming that sum, $>$, and $\in$ are primitives of $\mathcal{S}$, with the obvious meanings. It is a program for a particular case of the *clique problem*, namely finding a clique (or: maximally connected component) of an undirected graph (which is given through its integer-labeled vertex set V and its arc set A), such that the sum of its integer labels exceeds 22.

Example 2.2 Let us now look at the target language, namely clp(FD) [6]. Among the many possible forms of programs, there are the *global search* [23, 24] programs. They work as follows: after initialising a descriptor D to a tuple representing the space of all candidate solutions to problem X and possibly containing the empty partial solution, an auxiliary program rgs adds all the constraints on a representation Y' of the solution to the constraint store, and a solution Y to X is then generated from the constraint store and Y'; where rgs works by incrementally splitting the descriptor D by adding a solution element δ to its partial solution, yielding D', and constraining δ to achieve consistency with the partial solution so far, until no split is possible and a variablised representation Y' of the solution to X can be extracted from D. Formally, this dataflow and control-flow can be captured in the following open clp(FD) program for r [8]:

$$\begin{array}{rcll}
r(X,Y) & \leftarrow & initialise(X,D), & \\
 & & rgs(X,D,Y'), & \\
 & & generate(Y',Y,X) & \\
rgs(X,D,Y') & \leftarrow & extract(X,D,Y') & (gs_{dec}) \\
rgs(X,D,Y') & \leftarrow & split(D,X,D',\delta), & \\
 & & constrain(\delta,D,X), & \\
 & & rgs(X,D',Y') &
\end{array}$$

The only open symbols are relations $initialise$, $generate$, $extract$, $split$, and $constrain$. Note that $constrain$ just *posts* constraints on the search space, the actual solutions being *enumerated* by $generate$ once *all* constraints have been posted, because we use a constraint language.

The gs_{dec} program can be refined for subset decision problems, yielding the following (still) open program:

$$\begin{array}{rcl}
r(X,Y) & \leftarrow & initialise(X,D), \\
 & & rgs(X,D,Y'), \\
 & & generate(Y',Y,X) \\
rgs(X,D,Y') & \leftarrow & extract(X,D,Y') \\
rgs(X,D,Y') & \leftarrow & split(D,X,D',\delta), \\
 & & constrain(\delta,D,X), \\
 & & rgs(X,D',Y') \\
initialise(X,D) & \leftarrow & D = \langle X,[\,],[\,]\rangle \\
extract(\_,D,Y') & \leftarrow & D = \langle \{\},V,W\rangle, \\
 & & Y' = \langle V,W\rangle, \\
 & & g(Y') \\
split(D,X,D',\delta) & \leftarrow & D = \langle \{A|T\},V,W\rangle, \\
 & & B\ in\ 0..1, \\
 & & \delta = \langle A,B\rangle, \\
 & & D' = \langle T,[A|V],[B|W]\rangle \\
constrain(\_,D,\_) & \leftarrow & D = \langle \_,[\,],[\,]\rangle \\
constrain(\delta,D,X) & \leftarrow & \delta = \langle A,B\rangle, \\
 & & D = \langle \_,[E|V],[F|W]\rangle, \\
 & & B \#\wedge\ F \rightarrow p(A,E), \\
 & & constrain(\delta,\langle \_,V,W\rangle,X) \\
generate(Y',Y,\_) & \leftarrow & Y' = \langle \_,W\rangle, \\
 & & labeling([\,],W), \\
 & & int2ext(Y',Y) \\
int2ext(S_i,\{\}) & \leftarrow & S_i = \langle [\,],[\,]\rangle \\
int2ext(S_i,\{A|S_e\}) & \leftarrow & S_i = \langle [A|V],[1|W]\rangle, \\
 & & int2ext(\langle V,W\rangle,S_e) \\
int2ext(S_i,S_e) & \leftarrow & S_i = \langle [A|V],[0|W]\rangle, \\
 & & int2ext(\langle V,W\rangle,S_e) \\
E \in S & \leftarrow & ... \\
sum(S,N) & \leftarrow & ... \\
... & \leftarrow & ...
\end{array}$$

$$(gs^{subset}_{dec})$$

where in, $\#\wedge$, and $labeling$ are primitives of clp(FD) (with the obvious meanings). The implication $\rightarrow$ in the second clause for $constrain$ actually needs to be rewritten in a way more suitable for clp(FD), but explaining this goes beyond the scope of this paper, so we give this intuitive shorthand notation instead.

Descriptors take the form $\langle U,V,W\rangle$, where U is the subset (in external representation) of the given set T for whose elements it has not been decided yet whether they belong to the subset S or not, and $\langle V,W\rangle$ is the internal representation of the partially computed subset. The chosen representation in clp(FD), called the *internal representation*, of the subset S of the given finite set T (of n elements) is a mapping from T into Boolean values, that is we maintain n couples $\langle T_i,B_i\rangle$ where the (initially non-ground) Boolean B_i expresses whether the (initially ground) i^{th} element T_i of T is a member of S or not. The lists V and W in descriptors always contain a prefix of the n elements T_i and Booleans B_i, respectively. Note that we are thus restricted to finite sets. Also, the set T should not be too large, because the search space for S has size $O(2^n)$, whatever the size of S. Given this representation choice, it is easy to write *constraint-posting* clp(FD) programs for $\in$, sum, $int2ext$ (which converts between the internal and external representations), and all other classical set operations. An alternative representation of the subset S, namely as a sequence of k $(\leq n)$ variables constrained to be different elements of T, has two disadvantages compared to ours: first, the search space for S then is much worse, namely $O(n!)$, and second, an explicit loop for k ranging from 0 to n has to be wrapped around the code.

The only open symbols now are relations g and p. Note that they here operate on the internal representation of the subset, whereas they operate on the external one in the input $\mathcal{S}$ program. This assumes that each $\mathcal{S}$ primitive on sets has a corresponding clp(FD) program with the *same* name and parameters.

A constraint program for the considered particular case of the clique problem is a refinement of gs^{subset}_{dec} under the extension σ of Example 2.1. It is subject to obvious optimisations through partial evaluation, and should thus first be run through a partial evaluator, e.g., MIXTUS [21].

3 Program Schemas

We can now define program schemas, which are intended to represent entire families of similar programs. Contrary to programs, which are syntactic entities, program schemas are semantic entities, as we also need a semantic notion of what it means for a program to be a refinement of a program schema. Therefore, a program schema consists of an open program *and* a set of axioms, whose role is to prevent some programs from being undesired refinements of that open program.

Definition 3.1 A *program schema* is a couple $\langle T,A\rangle$, where *template* T is an open program, and *axioms* A are open formulas constraining the refinements of T.
Program P is a *refinement* of program schema $\langle T,A\rangle$ if P is a refinement of T under some extension θ, provided the 'definitions' in P 'satisfy' the axioms A.

A template captures the problem-independent dataflow and control-flow of an entire family of programs, whereas some refinement thereof (such that the axioms are 'satisfied') captures the problem-dependent computations of a member of that family.

Example 3.1 Let *Iff* $= \langle$ *iff* $, \emptyset\rangle$ denote the $\mathcal{S}$ program schema obtained from template *iff* (of Example 2.1) and the empty set of axioms, as we do not wish to impose any conditions on the (open) relations i_r and o_r.

Example 3.2 Let $Subset_{dec} = \langle subset_{dec}, \emptyset\rangle$ denote the $\mathcal{S}$ program schema obtained from template $subset_{dec}$ (also of Example 2.1) and the empty set of axioms, as we also do not wish to impose any conditions on the (open) relations g and p.

Example 3.3 According to our informal description of how global search programs work, the open relation symbols $initialise$, $generate$, $extract$, $split$, and $constrain$ of gs_{dec} can be informally specified as follows:

- $initialise(X, D)$ iff D is the descriptor of the initial space of candidate solutions to problem X;
- $extract(X, D, Y')$ iff the variablised internal representation Y' of a solution to problem X is directly extracted from descriptor D;
- $split(D, X, D', \delta)$ iff descriptor D' describes a subspace of D wrt problem X, such that D' is obtained by adding δ to descriptor D;
- $constrain(\delta, D, X)$ iff adding δ to descriptor D leads to a descriptor defining a sub-space of D that may contain correct solutions to problem X;
- $generate(Y', Y, X)$ iff correct solution Y to problem X corresponds to the internal representation Y' of a solution that is generated from the constraint store, which is an implicit parameter representing X.

It can be shown that their 'definitions' in refinement gs^{subset}_{dec} (of Example 2.2) indeed 'satisfy' these specifications.

Formalising these specifications is done through axioms. Let $\mathcal{Y}'$ be the type of internal representations of solutions, $\mathcal{D}$ be the type of search space descriptors, and Δ be the type of the elements of the partial solutions stored in descriptors. A first axiom expresses that $generate$ and r are equivalent when the constraint store is set up:

$$r(X, Y) \leftrightarrow \exists Y' : \mathcal{Y}' \,.\, generate(Y', Y, X) \qquad (A_0)$$

The next axiom expresses that all correct solutions Y to problem X are contained in the initial space for X:

$$\begin{gathered} r(X, Y) \rightarrow \exists D : \mathcal{D} \,.\, \exists Y' : \mathcal{Y}' \,. \\ initialise(X, D) \wedge int2ext(Y', Y) \wedge satisfies(Y', D) \end{gathered} \qquad (A_1)$$

where $satisfies(Y', D)$ means that the internal representation Y' of a solution is in the space described by descriptor D, which is the case if Y' can be extracted after a finite number of applications of $split$ to D. Formally:

$$\begin{aligned} & satisfies(Y', D) \leftrightarrow \exists k : int \,.\, \exists D' : \mathcal{D} \,.\, \exists \delta : \Delta \,. \\ & \quad split^k(D, X, D', \delta) \wedge extract(X, D', Y') \\ & \text{where :} \\ & \quad split^0(D, X, D', \delta) \leftrightarrow D = D' \\ & \text{and, for all } k : int : \\ & \quad split^{k+1}(D, X, D', \delta) \leftrightarrow \exists D'' : \mathcal{D} \,.\, \exists \delta' : \Delta \,. \\ & \qquad split(D, X, D'', \delta') \wedge split^k(D'', X, D', \delta) \end{aligned} \qquad (A_2)$$

Finally, we use constraint satisfaction to prune off branches of the search tree that cannot yield solutions. Given a space described by D and a (possibly still variablised) solution Y to problem X, if splitting D into D' makes D' contain the solution Y, then $constrain$ must succeed. Formally:

$$\begin{gathered} r(X, Y) \wedge int2ext(Y', Y) \wedge split(D, X, D', \delta) \\ \wedge\, satisfies(Y', D') \rightarrow constrain(\delta, D, X) \end{gathered} \qquad (A_3)$$

Conversely, the contrapositive of A_3 shows that if $constrain$ fails, then the new space described by D' (which is D plus δ) does not contain any solution to X. CP languages contain the SAT decision procedure, checking whether a constraint store is satisfiable [19].

This last axiom sets up a necessary condition that $constrain$ must establish. Given the left-hand side of the implication, such a condition can be derived using automated theorem proving (ATP), as shown in [22, 23]. Of course, we are not interested in too weak such a condition, such as the trivial solution $true$, but rather in a stronger one. However, deriving the absolutely strongest one (which establishes equivalence rather than implication) is impractical, because finding it may take too much time or may even turn out to be beyond current ATP possibilities, and because such a perfect $constrain$ would be too expensive to evaluate (since it would eliminate all backtracking in the solution generation). So we should (automatically, if possible) derive the strongest "possible and reasonable" condition, the criteria for these qualities being rather subjective. Fortunately, for many families of search problems, it turns out that this condition can be easily manually pre-computed at schema-design time, so that ATP technology is then unnecessary at compilation time. The refinement gs^{subset}_{dec} (of Example 2.2) shows how we did this for the family of subset decision problems.

Let $GS_{dec} = \langle gs_{dec}, \{A_0, A_1, A_2, A_3\}\rangle$ denote the clp(FD) program schema obtained from template gs_{dec} (of Example 2.2) and the axioms above.

Let $GS^{subset}_{dec} = \langle gs^{subset}_{dec}, \emptyset\rangle$ denote the clp(FD) program schema obtained from template gs^{subset}_{dec} (also of Example 2.2) and the empty set of axioms, as we do not wish to impose any conditions on the (open) relations g and p.

4 Schema-Guided Compilation

We now introduce the notion of programming schemas, which are useful for guiding our compilations.

Definition 4.1 A *programming schema* is a triple $\langle D_1, E, D_2 \rangle$, where D_1 and D_2 are program schemas, and open formula E is the condition under which any refinement of D_2 under some extension θ is 'open-equivalent' to the corresponding refinement of D_1 under θ.

Note that programming schemas are thus also semantic entities. Indeed, there are two (semantic) program schemas D_1 and D_2 in each programming schema, and there is a (semantic) condition E in it that ensures that passing from a refinement of D_1 to a refinement of D_2 under the same extension is an 'open-equivalence'-preserving operation. (In this paper, no programming schema is shown where $E \neq true$; for examples thereof, see [9]. This feature is heavily used in schema-guided optimisation [3], which is similar to schema-guided compilation.)

Example 4.1 The triple $\langle$ *Iff*$, true, GS_{dec} \rangle$ is a programming schema, capturing the compilation into global search clp(FD) programs of iff-programs in $\mathcal{S}$. Note that the open symbols in its input and target program schemas are different.

Example 4.2 The triple $\langle Subset_{dec}, true, GS_{dec}^{subset} \rangle$ is a programming schema, capturing the compilation into global search clp(FD) programs of subset decision programs in $\mathcal{S}$. Note that the open symbols in its input and target program schemas are the same, namely g and p.

In theory, one could use programming schemas with GS_{dec} as target schema (such as the one in Example 4.1) in a way analogous to the way a programming schema with a divide-and-conquer target schema was used in [22, 7] to guide compilation. This means following a *strategy* of (a) reusing, from a component base, clp(FD) programs for **some** of the open relations of the clp(FD) template gs_{dec} ('satisfying' the axioms of course), (b) propagating their corresponding $\mathcal{S}$ programs across the axioms $A_0 - A_3$ to set up $\mathcal{S}$ programs for the remaining open relations, (c) calling a (schema-guided) compiler to generate clp(FD) programs from these new $\mathcal{S}$ programs, and (d) assembling the overall compiled clp(FD) program by concatenating the reused clp(FD) template, the reused clp(FD) programs, and the recursively compiled clp(FD) programs.

In practice, however, this puts heavy demands on ATP technology, and in particular this turns out much more difficult for the GS_{dec} target schema than for the divide-and-conquer one [23]. Fortunately, a very large percentage of global search programs falls into one of seven families identified by Smith [23], each being a refinement of the gs_{dec} template where programs for **all** its open relations are chosen in advance so as to 'satisfy' axioms $A_0 - A_3$. These refinements are then still open, though no longer in the problem-independent relations of the target gs_{dec} template, but now in the problem-dependent relations of the input $\mathcal{S}$ template (such as in Example 4.2). We here investigate the family of subset decision problems. Elsewhere, we have already discussed how to do this for the families of assignment problems (where a mapping between two given sets has to be found, subject to some constraints) and permutation problems (where a sequence representing a permutation of a given set has to be found, subject to some constraints) [8], and binary split problems (where an integer has to be found within an ordered sequence, subject to some constraints) [14]. The remaining family identified by Smith (two of his seven families actually are particular cases of other ones), namely finding sequences (of given or bounded size) over a given set, has already been tackled. We have also discovered a useful generalisation of the family discussed in this paper, namely finding k subsets of a given set [13].

A particular case of schema-guided compilation is thus apparent now: given a program P in $\mathcal{S}$ and a programming schema $\langle D_1, E, D_2 \rangle$ where the open symbols in D_1 and D_2 are the same, find an extension θ under which P refines D_1, verify the equivalence condition (i.e., whether $\theta \vdash E$), and return the refinement $D_2 \cup \theta$, which is in clp(FD). In other words, *schema-guided compilation then simply amounts to replacing the template part of a refinement by another template, if the equivalence condition holds*. Figure 1 gives a representation of this process. We call this *schema-guided compilation through pre-computation* because the compilation act is carefully pre-computed, by hand and off-line, for the problem-independent part of an entire family of programs. Notice the (superficial) analogy with the Tetris game: to compile (win), it suffices to find for each arriving program (block) a program schema (shape) of which it is a refinement under some extension (into which it fits upon some rotation); the corresponding compiled program then is a refinement of the 'open-equivalent' program schema under the same extension (by applying the same rotation to that shape), but the analogy has already ended here. Also note that this technique is indeed independent of the chosen target language clp(FD).

Obviously, the resulting program can be fed into a schema-guided optimiser [3] (a tool that transforms a program into some other program of the *same* language, but that is more efficient, in space or time), yielding an iterative programming process that stops whenever no suitable programming schema can be found to continue. There is thus a smooth unification, and hence integration, of program compilation and program optimisation.

5 Conclusion

In this progress report on our research, we have introduced a novel approach to the compilation of very expressive, purely declarative, typed, first-order set constraint logic programs. While awaiting (more) efficient set constraint solvers, we have decided to exploit the well-known efficiency of finite-domain solvers and to compile (or: reformulate) set constraints into finite-domain constraints. Our approach is independent of the target language and will resist the trial of time, as we just have to rewrite the refinements in upcoming better target languages, as long as there is space for improvement along the declarative-

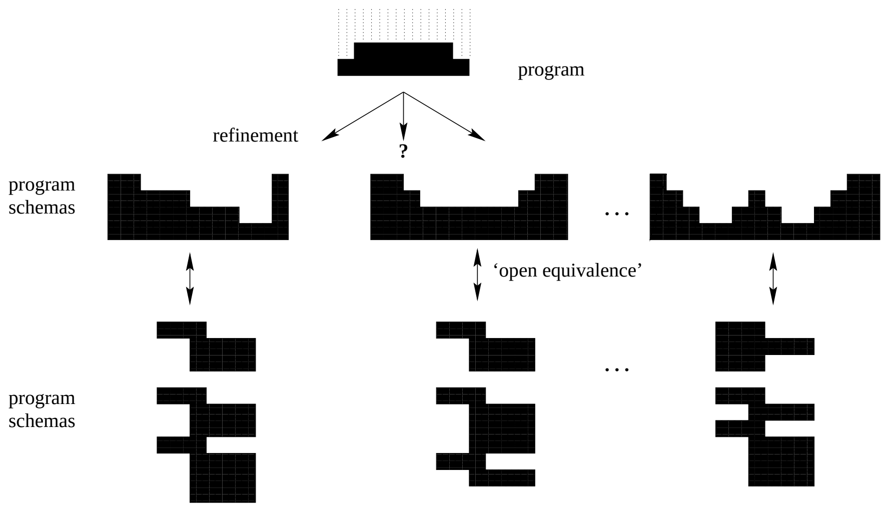

Figure 1: Schema-guided compilation through pre-computation for program schemas

ness and expressiveness axes.

We are ready to sacrifice any general-purpose nature of our input language, by just developing programming schemas for some problem families, if this is what it takes to facilitate (and speed up) constraint program development and to speed up the compiled programs. This philosophy is in line with the current trend of domain-specific tools, such as the PLANWARE [2] system for planning problems, or the domain-specific primitives of OPL (Optimisation Programming Language) [28] for scheduling and resource allocation problems.

5.1 Related Work

This work is inspired by D.R. Smith's research on synthesising global search programs (in *Refine*) from first-order logic specifications (also in *Refine*) with KIDS [23, 24] and its successor DESIGNWARE [25]. Our work concentrates on generating constraint programs instead. We thus only have to generate code that (incrementally) *posts* the constraints, because the actual constraint propagation and pruning are performed by the CP system. We have thus detached the problem-specific model (constraint part + search part) from the problem-independent solver. This allows us to generate only the model and to re-use the solver, whereas DESIGNWARE synthesises both. This required a significant re-engineering [8] of the original global search schema, so that it reflects a *constrain-and-generate* programming methodology.

Tests [8] have shown that at least one order of magnitude is gained in efficiency (before optimisation) by switching from an "ordinary" symbolic language, such as *Refine*, to a constraint one. In addition, our generated clp(FD) programs behave much more gracefully when the problem size increases, rather than seeing their run-times degenerate with problem size. These tests also showed that our automatically generated clp(FD) programs are only 3 to 5 times slower than carefully hand-crafted, published clp(FD) programs, which is encouraging since no optimisations are performed yet on our programs. Since our compilation is fully automatic, starting from short and elegant programs, our approach seems viable.

The other novelty compared to the DESIGNWARE approach is that we advocate that program schemas can be of (much) lower granularity than those of Smith (i.e., our templates can be refinements of his templates), so that the selection of the most appropriate programming schema, via the notion of refinement, can be done through rather trivial theorem proving, such as by performing matchings. Moreover, our differentiation between several program schemas where Smith only considers a single one allows the manual, off-line pre-computation, at schema design time, of more details of the corresponding equivalent program schema (such as the filters of global search), which can otherwise only be found, at compilation time, through sophisticated theorem proving. Hence we facilitated (and speeded up) compilation and could even further speed up the synthesised programs.

We thus try to cross-fertilise the results of the best two, but unfortunately so far orthogonal, approaches to optimisation problems, namely CP on the programming languages side and DESIGNWARE/PLANWARE on the program synthesis & transformation side. This cross-fertilisation should unite these approaches into a coherent whole, and reveal beneficial to both of them. Indeed, CP can benefit from the schemas plus theorem-proving approach of DESIGNWARE so as to make even higher-level languages possible, and DESIGNWARE can benefit from the re-use of a constraint solver rather than having to first synthesise a solver and then transform it (in user-guided fashion).

Other than inherently being a computationally incomplete language (by being limited to a finite set of problem families), our input language compares as follows to other languages. Compared to the CLPS [1], COJUNTO [11], and NP-SPEC [4] set constraint languages, our input language is more expressive (by not being limited to sets of initially known size), and by design much faster. The OZ [26] language also allows sets of a priori unknown size [20], and is faster than ours, but it is less expressive and less declarative. The OPL [28] constraint language sets new standards in expressiveness, but is currently limited to membership constraints with ground sets (of known size). The DESIGNWARE system [25] maybe allows the formulation of constraints on possibly infinite sets and on sets whose elements are drawn from an infinite domain, but its fully automated synthesis sub-system can only generate very slow programs (though its user-guided transformation sub-system can optimise them into extremely fast code; see below for more on this topic).

Our work can also be seen as being of methodological nature. Indeed, the identification of problem families and of efficient corresponding programming schemas is a contribution to constraint programming methodology. This is not unlike what is advocated with *design patterns* [10] (except that we aim at full formalisation and automation, whereas patterns are mostly informal) and *case-based reasoning* (CBR) [18]. Moreover, unlike the top-down decomposition methodology, for instance, which is solution-centered, our methodology is problem-centered and thus quite useful.

Very few works deal with the (manual or assisted) development of constraint programs. A manual methodology for developing constraint programs from informal specifications is given in [5]. In [16], the possibility of generating steadfast constraint programs is discussed, without exhibiting a method, though. The probably first work on automatically generating constraint programs from higher-level descriptions is [12], regarding the family of assignment problems. In [27], a language-independent computer-assisted constraint programming architecture is proposed, very much in line with our objectives, but no technical details on the compilation technique are given.

5.2 Future Work

Our plan now is to complete our investigation on adapting Smith's work to the CP paradigm, and then to port the results to the very recent OPL [28], which is much more expressive, but not (significantly) faster or more declarative than clp(FD). We have already detected expressiveness and declarativeness gaps in OPL, so we can continue to deploy our approach.

Our improvements on compilation over DESIGNWARE have not obliterated the need for *optimisation* of our compiled programs, as they are less efficient than the optimised ones that *can* result from *user-guided* DESIGNWARE optimisations. However, we wish to achieve *automated* optimisation of the compiled program. There are at least four approaches to the optimisation of (our compiled) constraint programs.

First, constraint programs (i.e., models) use the solver of the underlying CP language like a black box. One could thus specialise that solver for each particular model, for instance through partial evaluation. Specialisation can often be performed automatically and could then be linked back-to-back with the compiler. This avenue may fail, as current specialisation technology might not be powerful enough to achieve significant speed-ups of constraint programs, as opposed to the impressive results with more conventional declarative languages. Indeed, the style of the code of current solvers can probably not be changed, but this style is (currently) hardly amenable to specialisation.

However, it is interesting to note that DESIGNWARE optimisations *do* achieve a (user-guided) specialisation of the (synthesised) solver for the (synthesised) model. That specialisation is feasible because the synthesised solver really is a very high-level one compared to the ones we re-use. That high level is the key to the feasibility of such optimisation through specialisation, and the key to the *possibility* that such optimisation yields a more efficient (problem-specific) solver than a re-used (general-purpose) one. This is why the DESIGNWARE team does not want to re-use standard solvers. The "price" to pay is that their optimisations are user-guided and thus *not* guaranteed to lead to solvers that are as fast as re-used ones (unless domain-specific synthesisers, such as PLANWARE, are developed, but this is already a concession that some form of re-use is inevitable).

Second, so far, we only generate the constraint part of the model, but we leave the search part empty, hence relying on the default search procedure. One way to optimise the model thus is the *generation of a problem-specific search part*, such that the underlying (re-used) solver is better exploited than by the default search procedure. This is a virtually unexplored research area, at least as far as machine support is concerned. We plan to deploy DESIGNWARE-style compilation techniques towards this, using schemas and setting up lightweight theorem-proving obligations in order to reduce and organise the programming search space. We believe that this approach has the highest potential, eclipsing any possible speed-ups through specialisation of the solver, whether it is re-used or generated.

Third, *adding redundant constraints* often accelerates the solver. Such redundant constraints usually express properties of the problem that follow from the other constraints but that the solver cannot infer by itself and hence cannot use. This is again a little-explored field, at least as far as machine support is concerned, and we hope to innovate here as well.

Fourth, it has been noted that successful optimisation processes often follow some standard pattern [24]. Hence the idea that such processes can be precompiled into one-step macroscopic optimisation methods, also represented by *programming schemas*, hence providing a seamless integration of schema-guided compilation and optimisation. We have already exploited this idea for the optimisation of divide-and-conquer programs [3] and now plan to extend it to (the constraint and search parts of) constraint programs.

Acknowledgements

We thank Hamza Zidoum (UAE University), for insightful ideas during the incubation time for this research.

References

[1] F. Ambert, B. Legeard, and E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *Techniques et Sciences Informatiques* 15(3):297–328, 1996.

[2] L. Blaine, L. Gilham, J. Liu, D.R. Smith, and S. Westfold. PLANWARE: Domain-specific synthesis of high-performance schedulers. In D.F. Redmiles and B. Nuseibeh (eds), *Proc. of ASE'98*, pp. 270–279. IEEE Computer Society Press, 1998.

[3] H. Büyükyıldız and P. Flener. Generalised logic program transformation schemas. In N.E. Fuchs (ed), *Proc. of LOPSTR'97*, pp. 46–65. LNCS 1463. Springer-Verlag, 1998.

[4] M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In P. Flener and K.-K. Lau (eds), *Pre-Proc. of LOPSTR'98*, pp. 1–9. TR UMCS-98-6-1, Univ. of Manchester, 1998. (ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-98-6-1.html)

[5] Y. Deville and P. Van Hentenryck. Construction of CLP programs. In D.R. Brough (ed), *Logic Programming: New Frontiers*, pp. 112–135, Kluwer, 1992.

[6] D. Diaz and Ph. Codognet. A minimal extension of the WAM for clp(FD). In D.S. Warren (ed), *Proc. of ICLP'93*, pp. 774–790. The MIT Press, 1993.

[7] P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In M. Lowry and Y. Ledru (eds), *Proc. of ASE'97*, pp. 153–160. IEEE Computer Society Press, 1997.

[8] P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. In D.F. Redmiles and B. Nuseibeh (eds), *Proc. of ASE'98*, pp. 168–176. IEEE Computer Society Press, 1998.

[9] P. Flener and J. Richardson. A unified view of programming schemas and proof methods. In A. Bossi (ed), *Pre-Proc. of LOPSTR'99*. TR, University of Venice, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.

[12] T. Hjerpe. *High-Level Specification and Efficient Solving of Constraint Satisfaction Problems*. Ph.D. Thesis, Uppsala University, 1995.

[13] B. Hnich and Z. Kızıltan. Generating programs for k-subsets problems. In P. Alexander (ed), *Proc. of the ASE'99 Doctoral Symposium*. 1999.

[14] Z. Kızıltan. *Schema-Guided Synthesis of Constraint Logic Programs*. Magister Thesis, Uppsala University, 1999.

[15] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D.S. Warren (ed), *Proc. of ICLP'93*, pp. 441–455. The MIT Press, 1993.

[16] K.-K. Lau and M. Ornaghi. A formal approach to deductive synthesis of constraint logic programs. In J.W. Lloyd (ed), *Proc. of ILPS'95*, pp. 543–557. The MIT Press, 1995.

[17] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. of Logic Programming* 38(3):259–294, March 1999.

[18] D. Leake. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI Press, 1996.

[19] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[20] T. Müller. Solving set partitioning problems with constraint programming. In *Proc. of PAPPACT'98*, pp. 313–332. The Practical Application Co., 1998.

[21] D. Sahlin. The MIXTUS approach to automatic partial evaluation of full Prolog. In *Proc. of NACLP'90*. The MIT Press, 1990.

[22] D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.

[23] D.R. Smith. The structure and design of global search algorithms. *TR KES.U.87.12*, Kestrel Institute, 1988.

[24] D.R. Smith. KIDS: A semi-automatic program development system. *IEEE Trans. on Software Engineering* 16(9):1024–1043, 1990.

[25] D.R. Smith. Toward a classification approach to design. *Proc. of AMAST'96*, pp. 62–84. LNCS 1101. Springer-Verlag, 1996.

[26] G. Smolka. The OZ programming model. In J. van Leeuwen (ed), *Computer Science Today*, pp. 324–343. LNCS 1000. Springer-Verlag, 1995.

[27] E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In J. Little (ed), *Proc. of PACLP'99*, pp. 81–93. The Practical Application Co., 1999.

[28] P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.