Exploiting functional dependencies in declarative problem specifications

Marco Cadoli and Toni Mancini

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
cadoli|tmancini@dis.uniroma1.it

Abstract. In this paper we tackle the issue of the automatic recognition of functional dependencies among guessed predicates in constraint problem specifications. Functional dependencies arise frequently in pure declarative specifications, because of the intermediate results that need to be computed in order to express some of the constraints, or due to precise modelling choices, e.g., to provide multiple viewpoints of the search space in order to increase propagation. In either way, the recognition of dependencies greatly helps solvers, letting them avoid spending search on unfruitful branches, while maintaining the highest degree of declarativeness. By modelling constraint problem specifications as second-order formulae, we provide a characterization of functional dependencies in terms of semantic properties of first-order ones. Additionally, we show how suitable search procedures can be automatically synthesized in order to exploit recognized dependencies. We present OPL examples of various problems, from bio-informatics, planning and resource allocation fields, and show how in many cases OPL greatly benefits from the addition of such search procedures.

1 Introduction

Reasoning on constraint problems in order to change their formulations has been proven to be of fundamental importance in order to speed-up the solving process. To this end, different approaches have been proposed in the literature, like symmetry detection and breaking (cf., e.g., [1, 8]), the addition of implied constraints (cf., e.g., [24]), and the use of redundant models, i.e., multiple viewpoints synchronized by channelling constraints, in order to increase constraint propagation [7, 26, 18]. Many of these approaches either are designed for a specific constraint problem, or act at the instance level, and very little work has been performed at the level of problem specification (also called "model"). Indeed, many of the properties of constraint problems amenable to optimizations, e.g., symmetries, existence of "useful" implied constraints, or of multiple viewpoints, strongly depend on the problem structure, rather than on the particular input instance considered. Hence, their recognition at the instance level, where the problem structure has been almost completely hidden, may easily become not convenient and expensive.

Moreover, almost all state-of-the-art systems for constraint modelling and programming (e.g., AMPL [14], OPL [25], DLV [12], SMODELS [22], and NP-SPEC [5])

exhibit a clear separation between problem specification and input instances, allowing the user to focus on the declarative aspects of the problem model before instantiation, and without committing *a priori* to a specific solver. In fact, some systems, e.g., AMPL [14], are able to translate –at the request of the user– a specification in various formats, suitable for different solvers.

Taking advantage of this separation, the goal of our research is to detect and exploit those properties of constraint problems amenable to optimizations that derive from the problem structure, at the symbolic level, before binding the specification to a particular instance, thus leading to the automatic reformulation of specifications.

In related work, we tackle the issues of highlighting some of the constraints of a specification that can be ignored in a first step, and then efficiently reinforced (i.e., without performing additional search, the so-called "safe delay" constraints) [3], and that of detecting structural (i.e., problem-dependent) symmetries, and breaking them by adding symmetry-breaking constraints to the problem specification [2]. We additionally show that these tasks can be performed automatically by computer tools, e.g., first-order theorem provers or finite model finders [4].

In this paper we exploit another interesting property of constraint problems, i.e., the functional dependencies that can hold among predicates in problem specifications. Informally, given a problem specification, a predicate is said to be functional dependent on the others if, for every solution of any instance, its extension is determined by the extensions of the others.

The presence of functional dependencies is common in problem specifications for different reasons: as an example, to allow the user to have multiple views of the search space, in order to be able to express the various constraints under the most convenient viewpoint, or to maintain aggregate or intermediate results needed by some of the constraints, as the following example shows.

Example 1 (The HP 2D-Protein folding problem [20]). This problem specification models a simplified version of one of the most important problems in computational biology. It consists in finding the spatial conformation of a protein (i.e., a sequence of amino-acids) with minimal energy.

The simplifications with respect to the real problem are twofold: firstly, the 20-letter alphabet of amino-acids is reduced to a two-letter alphabet, namely H and P. H represents *hydrophobic* amino-acids, whereas P represents polar or *hydrophilic* amino-acids. Secondly, the conformation of the protein is limited to a bi-dimensional discrete space. Nonetheless, these limitations have been proven to be very useful for attacking the whole protein conformation prediction protein, which is known to be NP-complete [9] and very hard to solve in practice.

In this formulation, given the sequence (of length n) of amino-acids of the protein (the so called primary structure of the protein), i.e., a sequence of length n with elements in {H,P}, we aim to find a connected shape of this sequence on a bi-dimensional grid (which points have coordinates in the integral range $\texttt{Coord} = [-(n-1),(n-1)]$, the sequence starting at $(0,0)$), which is not overlapping, and maximizes the number of "contacts", i.e., the number of non-sequential pairs of H amino-acids for which the Euclidean distance of the positions is 1 (the overall energy of the protein is defined as the opposite of the number of contacts).

Different alternatives for the search space obviously exist: as an example, we can guess the position on the grid of each amino-acid in the sequence, and then force the obtained shape to be connected, non-crossing, and with minimal energy. A second approach is to guess the shape of the protein as a connected path starting at $(0, 0)$, by guessing, for each position t of the sequence, the direction that the amino-acid at the t-th position in the sequence assumes with respect to the previous one (directions in the HP-2D model can only be North, South, East, West). It is easy to show that the former model would lead to a search space of $(2n)^{2n}$ points, while the latter to a much smaller one (4^n points).[1]

However, choosing the latter model is not completely satisfactory. In fact, to compute the number of contacts in the objective function, absolute coordinates of each amino-acid in the sequence must be computed and maintained. It is easy to show that these values are completely defined by (i.e., functionally dependent on) the sequence of directions taken by the protein. □

Given a problem like the HP 2D-Protein folding one, if writing a procedural program, e.g., in C++, to solve it, possibly using available libraries for constraint programming, a smart programmer would avoid predicates encoding absolute coordinates of amino-acids to be part of the search space. Extensions for these predicates instead, would be *computed* starting from extensions of the others.

On the other hand, when using a declarative language for constraint modelling like, e.g., OPL, the user looses the power to distinguish among predicates which extension has to be found through a true search, from those which can be computed starting from the others, since all of them become of the same nature. Hence, the search space actually explored by the system can be ineffectively much larger, and additional information should be required from the user to distinguish among them, thus greatly reducing the declarativeness of the specification. To this end, the ability of the system to *automatically recognize* whether a predicate is functionally dependent on (or defined from) the others becomes of great importance from an efficiency point of view, since it can lead to significant reductions of the search space, although retaining the highest level of declarativeness.

The technique of avoiding branches on dependent predicates has already been successfully applied at the instance level for solving, e.g., SAT instances. As an example, it is shown in [16] how to modify the Davis-Putnam procedure for SAT so that it avoids branches on variables added during the clausification of non-CNF formulae, since values assigned to these variables depend on assignments to the other ones. Moreover, some SAT solvers, e.g., EQSATZ [21], have been developed in order to appropriately handle (by means of the so-called "equivalence reasoning") equivalence clauses, which have been recognized to be a very common structure in the SAT encoding of many hard real-world problems, and a major obstacle to the Davis-Putnam procedure.

We believe that looking for dependent predicates at the specification level, rather than after instantiation, can be much more natural, since these issues strongly depend on the structure of the problem. To this end, our approach is to

[1] Actually, as for the second model, possible directions of each amino-acid with respect to the previous one can be only three, because of the non-crossing constraint. Nonetheless, we opt for the simpler model to enhance readability.

give a formal characterization of functional dependencies suitable to be checked by computer tools, and ultimately, to transform the original problem specification by automatically adding an explicit search strategy that exploits such dependencies, avoiding branches on dependent predicates. Moreover, in those cases in which functional dependencies derive from the adoption of multiple viewpoints of the search space, we could choose the best maximal set of independent predicates to branch on, depending on the amenability of the relevant constraints to propagation (cf., e.g, [18]).

2 Preliminaries

In this paper, we use *existential second-order logic* (ESO) for the formal specification of problems, which allows to represent all search problems in the complexity class NP [13]. Actually, constraint modelling systems like those mentioned in Section 1 have a richer syntax and more complex constructs. However, all of them are extensions of ESO on finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the constraint modelling paradigm. Yet, examples using the syntax of the implemented language OPL are shown in Section 4. We observe that these examples present also arithmetic constraints.

Coherently with all state-of-the-art systems, we represent an instance of a problem by means of a *relational database*. All constants appearing in a database are *uninterpreted*, i.e., they don't have a specific meaning.

An ESO specification describing a search problem π is a formula ψ_π:

$$\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R}), \tag{1}$$

where $\boldsymbol{R} = \{R_1, \ldots, R_k\}$ is the relational schema for every input instance (i.e., a fixed set of relations or given arities denoting the schema for all input instances for π), and ϕ is a quantified first-order formula on the relational vocabulary $\boldsymbol{S} \cup \boldsymbol{R} \cup \{=\}$ ("=" is always interpreted as identity).

An instance $\mathcal{I}$ to the problem is given as a relational database over the schema $\boldsymbol{R}$, i.e., as an extension for all relations in $\boldsymbol{R}$. Predicates (of given arities) in the set $\boldsymbol{S} = \{S_1, \ldots, S_n\}$ are called *guessed*, and their possible extensions (with tuples on the domain given by constants occurring in $\mathcal{I}$ plus those occurring in ϕ, i.e., the so called Herbrand universe) encode points in the search space for problem π. Formula ψ_π correctly encodes problem π if, for every input instance $\mathcal{I}$, a bijective mapping exists between solutions to $\pi(\mathcal{I})$ and extensions of predicates in $\boldsymbol{S}$ which verify $\phi(\boldsymbol{S}, \mathcal{I})$. It is worthwhile to note that, when a specification is instantiated against an input database, a CSP in the sense of [10] is obtained.

Example 2 (Graph 3-coloring [15, Prob. GT4]). In this NP-complete decision problem the input is a graph, and the question is whether it is possible to give each of its nodes one out of three colors (red, green, and blue), in such a way that adjacent nodes (not including self-loops) are never colored the same way. The question can be easily specified as an ESO formula ψ on the input schema $\boldsymbol{R} = \{edge(\cdot, \cdot)\}$:

$$\exists RGB \qquad \forall X \quad R(X) \vee G(X) \vee B(X) \ \wedge \tag{2}$$

$$\forall X \quad R(X) \rightarrow \neg G(X) \ \wedge \tag{3}$$
$$\forall X \quad R(X) \rightarrow \neg B(X) \ \wedge \tag{4}$$
$$\forall X \quad B(X) \rightarrow \neg G(X) \ \wedge \tag{5}$$
$$\forall XY \ X \neq Y \wedge R(X) \wedge R(Y) \rightarrow \neg edge(X,Y) \wedge \tag{6}$$
$$\forall XY \ X \neq Y \wedge G(X) \wedge G(Y) \rightarrow \neg edge(X,Y) \wedge \tag{7}$$
$$\forall XY \ X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X,Y). \tag{8}$$

Clauses (2) and (3-5) force every node to be assigned exactly one color (covering and disjointness constraints), while (6-8) force nodes linked by an edge to be assigned different colors (good coloring constraints). □

3 Definitions and formal results

In this section we give the formal definition of functionally dependent guessed predicate in a specification, and show how the problem of checking whether a set of guessed predicates is dependent from the others reduces to check semantic properties of a first-order formula.

Definition 1 (Functional dependence of a set of predicates in a specification). *Given a problem specification $\psi \doteq \exists \boldsymbol{SP}\ \phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$, with input schema $\boldsymbol{R}$,* $\boldsymbol{P}$ functionally depends *on $\boldsymbol{S}$ if, for each instance $\mathcal{I}$ of $\boldsymbol{R}$ and for each pair of interpretations M, N of $(\boldsymbol{S}, \boldsymbol{P})$ it holds that, if*

1. *$M \neq N$, and*
2. *$M, \mathcal{I} \models \phi$, and*
3. *$N, \mathcal{I} \models \phi$,*

then $M_{|\boldsymbol{S}} \neq N_{|\boldsymbol{S}}$, where $\cdot_{|\boldsymbol{S}}$ denotes the restriction of an interpretation to predicates in $\boldsymbol{S}$.

The above definition states that $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$, or that $\boldsymbol{S}$ functionally determines $\boldsymbol{P}$, if it is the case that, regardless of the instance, each pair of distinct solutions of ψ must differ on predicates in $\boldsymbol{S}$, which is equivalent to say that no two different solutions of ψ exist that coincide on the extension for predicates in $\boldsymbol{S}$ but differ on that for predicates in $\boldsymbol{P}$.

Example 3 (Graph 3-coloring, Example 2 continued). In the 3-coloring specification, one of the three guessed predicates is functionally dependent on the others. As an example, B functionally depends on R and G, since, regardless of the instance, it can be defined as $\forall X \ B(X) \leftrightarrow \neg(R(X) \vee G(X))$: constraint (2) is equivalent to $\forall X \ \neg(R(X) \vee G(X)) \rightarrow B(X)$ and (4) and (5) imply $\forall X \ B(X) \rightarrow \neg(R(X) \vee G(X))$. In other words, for every input instance, no two different solutions exist that coincide on the set of red and green nodes, but differ on the set of blue ones. □

Example 4 (Not-all-equal Sat [15, Prob. LO3]). In this NP-complete problem the input is a propositional formula in CNF, and the question is whether it is possible to assign a truth value to all the variables in such a way that the input formula is satisfied, and that every clause contains at least one literal whose truth value is false. We assume that the input formula is encoded by the following relations:

– $inclause(\cdot,\cdot)$; tuple $\langle l, c\rangle$ is in $inclause$ iff literal l is in clause c;
– $l^+(\cdot,\cdot)$ (resp. l^-); a tuple $\langle l, v\rangle$ is in l^+ (resp. l^-) iff l is the positive (resp. negative) literal relative to variable v, i.e., v itself (resp. $\neg v$);
– $var(\cdot)$, containing the set of propositional variables occurring in the formula;
– $clause(\cdot)$, containing the set of clauses of the formula.

A specification for this problem is as follows (T and F represent the set of variables whose truth value is true and false, respectively):

$$\exists TF\ \forall X\ var(X) \leftrightarrow T(X) \vee F(X)\ \ \wedge \tag{9}$$
$$\forall X\ \neg(T(X) \wedge F(X))\ \ \wedge \tag{10}$$
$$\forall C\ clause(C) \rightarrow$$
$$\big[\exists L\ inclause(L,C) \wedge \forall V\ \big(l^+(L,V) \rightarrow T(V)\big) \wedge \big(l^-(L,V) \rightarrow F(V)\big)\big]\ \wedge \tag{11}$$
$$\forall C\ clause(C) \rightarrow$$
$$\big[\exists L\ inclause(L,C) \wedge \forall V\ \big(l^+(L,V) \rightarrow F(V)\big) \wedge \big(l^-(L,V) \rightarrow T(V)\big)\big]. \tag{12}$$

Constraints (9–10) force every variable to be assigned exactly one truth value; moreover, (11) forces the assignment to be a model of the formula, while (12) leaves in every clause at least one literal whose truth value is false.

One of the two guessed predicates T and F is dependent on the other, since by constraints (9–10) it follows, e.g., $\forall X\ F(X) \leftrightarrow var(X) \wedge \neg T(X)$. □

It is worth noting that Definition 1 is strictly related to the concept of *Beth implicit definability*, well-known in logic (cf., e.g., [6]). We will further discuss this relationship in Section 5. In what follows, instead, we show that the problem of checking whether a subset of the guessed predicates in a specification is functionally dependent on the remaining ones, reduces to semantic properties of a first-order formula (proofs are omitted for lack of space). To simplify notations, given a list of predicates $\boldsymbol{T}$, we write $\boldsymbol{T'}$ for representing a list of predicates of the same size with, respectively, the same arities, that are fresh, i.e., do not occur elsewhere in the context at hand. Also, $\boldsymbol{T} \equiv \boldsymbol{T'}$ will be a shorthand for the formula

$$\bigwedge_{T \in \boldsymbol{T}} \forall \boldsymbol{X}\ T(\boldsymbol{X}) \equiv T'(\boldsymbol{X}),$$

where T and T' are corresponding predicates in $\boldsymbol{T}$ and $\boldsymbol{T'}$, respectively, and $\boldsymbol{X}$ is a list of variables of the appropriate arity.

Theorem 1. *Let $\psi \doteq \exists \boldsymbol{SP}\ \phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$ be a problem specification with input schema $\boldsymbol{R}$. $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$ if and only if the following formula is valid:*

$$[\phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R}) \wedge \phi(\boldsymbol{S'}, \boldsymbol{P'}, \boldsymbol{R}) \wedge \neg(\boldsymbol{SP} \equiv \boldsymbol{S'P'})] \rightarrow \neg(\boldsymbol{S} \equiv \boldsymbol{S'}). \tag{13}$$

Unfortunately, the problem of checking whether the set of predicates in $\boldsymbol{P}$ is functionally dependent on the set $\boldsymbol{S}$ is undecidable, as the following result shows:

Theorem 2. *Given a specification on input schema $\boldsymbol{R}$, and a partition $(\boldsymbol{S}, \boldsymbol{P})$ of its guessed predicates, the problem of checking whether $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$ is not decidable.*

Even if Theorem 2 states that checking functional dependencies in a specification is an undecidable task, in practical circumstances it can be effectively and often efficiently performed by automatic tools, such as first-order theorem provers and finite model finders (cf. [4] for details and some experiments).

4 Further examples

In this section we present some problem specifications which exhibit functional dependencies among guessed predicates. Due to their high complexity, we don't give their formulations as ESO formulae, but show their specifications in the well known language for constraint modelling OPL.

Example 5 (The HP 2D-Protein folding problem, Example 1 continued). As already stated in Example 1, rather than guessing the position on the grid of each amino-acid in the sequence, we chose to represent the shape of the protein by a guessed predicate `Moves[]`, that encodes, for each position t, the direction that the amino-acid at the t-th position in the sequence assumes with respect to the previous one (the sequence starts at $(0, 0)$). An OPL specification for this problem is shown in Section A.1. The analogous of the instance relational schema, guessed predicates, and constraints can be clearly distinguished in the OPL code.

To compute the number of contacts in the objective function (not shown in the OPL code for brevity), absolute coordinates of each amino-acid in the sequence must be calculated (predicates `X[]` and `Y[]`).

As for the non-overlapping constraint, an *all-different* constraint for the set of positions of all amino-acids in the sequence is needed. Unfortunately, OPL does not admit global all-different constraints to be stated on multi-dimensional arrays. So we decided to use an additional guessed predicate `Hits[]` in order to maintain, for every position on the grid, the number of amino-acid of the protein that are placed on it at each point during the construction of the shape (instead of using $\mathcal{O}(n^2)$ binary inequalities). For all of them, this number cannot be greater than 1, which implies that the string does not cross.

From the considerations above, it follows that guessed predicates `X[]`, `Y[]`, and `Hits[]`, are functionally dependent on `Moves[]`. □

Example 6 (The Sailco inventory problem [25, Section 9.4, Statement 9.17]). This problem specification, part of the OPLSTUDIO distribution package (as file `sailco.mod`), models a simple inventory application, in which the question is to decide how many sailboats the Sailco company has to produce over a given number of time periods, in order to satisfy the demand and to minimize production costs. The demand for the periods is known and, in addition, an `inventory` of boats is available initially. In each period, Sailco can produce a maximum number of boats (`capacity`) at a given unitary cost (`regularCost`). Additional boats can be produced, but at higher cost (`extraCost`). Storing boats in the inventory also has a cost per period (`inventoryCost` per boat).

Section A.2 shows an OPL model for this problem. From the specification it can be observed that the amount of boats in the inventory for each time period `t` > 0 (i.e., `inv[t]`) is defined in terms of the amount of regular and extra boats produced in period `t` by the following relationship: `inv[t] = regulBoat[t] + extraBoat[t] - demand[t] + inv[t-1]`. □

Example 7 (The Blocks world problem [23, 27]). In the Blocks world problem, the input consists of a set of blocks that are arranged in stacks on a table. Every block can be either on the table or onto another block. Given the initial and the desired configurations of the blocks, the problem amounts to find a minimal sequence of moves that allows to achieve the desired configuration. Every move is performed on a single clear block (i.e., on a block with no blocks on it) and moves it either onto a clear block or on the table (which can accommodate an arbitrary number of blocks). It is worth noting that a plan of length less than or equal to twice the number of blocks always exists, since original stacks can all be flattened on the table before building the desired configuration.

In our formulation, given in Section A.3, we assume that the input is given as an integer `nblocks`, i.e., the number of blocks, and functions `OnAtStart[]` and `OnAtGoal[]`, encoding, respectively, the initial and desired configuration. As for the guessed functions, `MoveBlock[]` and `MoveTo[]` respectively state, for each time point `t`, which block has been moved at time point `t-1`, and its new position at time `t`. Moreover, we use guessed functions `On[]`, that states the position (which can be either a block or the table) of a given block at a given time point, and `Clear[]`, that states whether a given block is clear at a time point. We observe that guessed functions `On[]` and `Clear[]` are functionally dependent on `MoveBlock[]` and `MoveTo[]`. □

5 Exploiting functional dependencies

Once a set $\boldsymbol{P}$ of guessed predicates of a problem specification $\psi \doteq \exists \boldsymbol{S}\boldsymbol{P}\ \phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$ has been recognized to be functionally dependent on the others, different approaches can be followed in order to exploit such a dependence: the most simple and elegant one is to force the system to branch only on defining predicates, i.e., those in $\boldsymbol{S}$, avoiding spending search on those in $\boldsymbol{P}$. An alternative approach is to substitute in the specification all occurrences of predicates in $\boldsymbol{P}$ with their definition, and we will briefly discuss it in the following.

As already observed in Section 3, the concept of functional dependence among guessed predicates expressed in Definition 1 is strictly related to the one of *Beth implicit definability* (cf., e.g, [6]). In particular, given a problem specification $\psi \doteq \exists \boldsymbol{S}\boldsymbol{P}\ \phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$, guessed predicates in set $\boldsymbol{P}$ functionally depend on those in $\boldsymbol{S}$ if and only if the first-order formula $\phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$ implicitly defines predicates in $\boldsymbol{P}$ (i.e., if every $\langle \boldsymbol{S}, \boldsymbol{R} \rangle$-structure has at most one expansion to a $\langle \boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R} \rangle$-structure satisfying $\phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$). It is worth remarking that, since we are interested in finite extensions for guessed predicates, Beth implicit definability has to be intended *in the finite*. Now, the question that arises is whether it is possible to derive, once a functional dependence (or, equivalently, an implicit definition) has been established, a formula that *explicitly defines* the dependent predicates in terms of the others. This formula, then, could take the place of all occurrences of those predicates in the problem specification. Although such a formula always exists in unrestricted first-order logic, this is not the case when only finite models are allowed. This is because first-order logic does not have the *Beth property in the finite* (cf., e.g., [11], and the intrinsically inductive definition of guessed function `inv[]` in Example 6). Nonetheless, in some cases such a formula indeed exists

(cf., e.g., Examples 3 and 4). On the other hand, a second-order explicit definition of a dependent predicate would not be adequate, since new quantified predicates have to be added to the specification, and, moreover, the obtained specification may not be in ESO any more. Hence, we follow the first approach, i.e., forcing the system to avoid branches on dependent predicates, in order to save search. We also observe that, ultimately, this achieves exactly the same goal the second alternative would reach: in both cases, the search space actually explored is the one defined by predicates in $\boldsymbol{S}$ only.

In this section we show how an explicit search strategy that avoids branches on dependent predicates can be automatically synthesized. To this end, we assume to face with languages that allow the user to provide an explicit search strategy, and, in particular, since its description may depend on the particular solver, we present examples by focusing on the constraint language OPL. OPL does not require a search strategy (called, in the OPL syntax, "search procedure") to be defined by the user, since it automatically uses default strategies when none is explicitly defined. On the other hand, it provides the designer with the possibility of explicitly programming in detail how to branch on variables, and how to split domains, by adding a `search` part in the problem specification.

The simplest way to provide a search procedure is by using the `generate()` construct, which receives a guessed predicate as input and forces OPL to generate all possible extensions for it, letting the policy for the generation to the system defaults. Of course, multiple occurrences of `generate()` (with different guessed predicates as arguments) are allowed. OPL also allows for more explicit search procedures, in which the programmer can explicitly choose the order for the generation of the various extensions for that predicate (cf. [25]). However, these topics require an accurate knowledge of the particular problem at hand, and are out of the scope of this paper, since they are less amenable to be generalized and automated.

Hence, given a problem specification in which a set $\boldsymbol{P}$ of the guessed predicates is functionally dependent on the others (set $\boldsymbol{S} = \{S1, \ldots, Sm\}$), a search procedure that forces OPL to avoid branches on predicates in $\boldsymbol{P}$ is the following:

```
search { generate(S1);  ...  generate(Sm); };
```

Search procedures added to the OPL specifications of HP 2D-Protein folding, Sailco inventory, and Blocks world problems in order to deal with the above described dependencies are, respectively, the following ones:

```
search { generate(Moves); };
search { generate(regulBoats);  generate(extraBoats); };
search { generate(MoveBlock);   generate(MoveTo); };
```

It is worth noting that, for some specifications, sets $\boldsymbol{S}$ and $\boldsymbol{P}$ are interchangeable. This intuitively happens when the user adopts multiple viewpoints of the search space (cf., e.g., Example 5, in which set $\{X, Y\}$ depends on $\{Moves\}$ and vice versa). In those cases, a first choice for deciding which set should be regarded as "defining" (i.e., $\boldsymbol{S}$), may involve the size of the associated search space.

To test the effectiveness of adding such search procedures, we made an experimentation of the above problems with OPL. In particular, we made the following experiments:

Length	Time		Saving %
	w/out search proc.	with search proc.	
14	–	1	99.9
30	–	6	99.9
36	3078	13	85.1
37	3386	17	93.6
45	–	99	>97.3
48	–	156	>95.7
50	–	215	>94.0
57	–	69	>98.0
60	–	128	>96.4
64	–	233	>93.5
102	1242	1007	18.9
123	914	913	0.0
136	761	760	0.0

Table 1. OPL solving times for some of the 44 benchmark instances of the HP 2D-Protein folding problem, with and without search procedure. ('–' means that OPL did not terminate in one hour.)

Instance	Blocks	Minimal plan length	Time		Saving %
			w/out search proc.	with search proc.	
bw-sussman	3	3	12	<1	>91.7
bw-reversal4	4	4	1	1	∼0
bw-4.1.1	4	6	882	<1	>99.9
bw-4.1.2	4	5	1879	<1	>99.9
bw-5.1.1	5	8	–	1	>99.9
bw-5.1.3	5	7	935	27	97.1
bw-large-a	9	12	–	–	–

Table 2. OPL solving times for some benchmark instances of the Blocks world problem, with and without search procedure. ('–' means that OPL did not terminate in one hour.)

- The HP 2D-Protein folding problem, on 44 instances taken from [17];
- The Blocks world problem, on structured instances, some of them used as benchmarks in [19];
- The Sailco inventory problem on random instances.

For both HP 2D-Protein folding and Blocks world, adding a search procedure that explicitly avoids branches on dependent predicates always speeds-up the computation, and very often the saving in time in very impressive. Tables 1 and 2 report typical behaviors of the system. On the other hand, for what concerns the Sailco inventory problem, no saving in time has been observed. This can be explained by observing that the problem specification is linear, and the linear solver automatically chosen by the system (i.e., CPLEX) is built on different technologies (e.g., the simplex method) that are not amenable to this kind of program transformation.

6 Conclusions

In this paper we discussed a semantic logical characterization of functional dependencies among guessed predicates in declarative constraint problem specifications. Functional dependencies can be very easily introduced during declarative modelling, either because intermediate results have to be maintained in order to express some of the constraints, or because of precise choices, e.g., redundant modelling. However, dependencies can negatively affect the efficiency of the solver, since the search space can become much larger, and additional information from the user is today needed in order to efficiently cope with them.

We described how, in our framework, functional dependencies can be checked by computer, and can lead to the automated synthesis of search strategies that avoid the system spending search in unfruitful branches. Several examples of constraint problems that exhibit dependencies have been presented, from bioinformatics, planning, and resource allocation fields, and experimental results have been discussed, showing that current systems for constraint programming greatly benefit from the addition of such search strategies.

Appendix: OPL code for the examples

A.1 The HP 2D-Protein folding problem

```
int+ n = ...;                                  // Part of the inst. schema: string length
enum Aminoacid {H,P};
range Pos [0..n-1];
range PosButLast [0..n-2];
Aminoacid seq[Pos] = ...;                      // Part of the inst. schema: amino-acid seq.
enum Dir {N,E,S,W};
range Coord [-(n-1)..n-1];
range Hit [0..n/2];

// Guessed predicates
var Dir Moves[PosButLast];                     // Protein shape
var Coord X[Pos], Y[Pos];                      // Absolute coordinates
var Hit Hits[Coord,Coord,Pos];

maximize ...                                   // Objective function (omitted)
subject to {
   X[0] = 0;  Y[0] = 0;                        // Pos. of initial elem. of the seq.
   forall (t in Pos: t > 0) {                  // Chann. constr's for position
       Moves[t-1] = N => (X[t] = X[t-1]     & Y[t] = Y[t-1] + 1);
       Moves[t-1] = S => (X[t] = X[t-1]     & Y[t] = Y[t-1] - 1);
       Moves[t-1] = E => (X[t] = X[t-1] + 1 & Y[t] = Y[t-1]);
       Moves[t-1] = W => (X[t] = X[t-1] - 1 & Y[t] = Y[t-1]);
   };
   forall (x,y in Coord : x<>0 \/ y<>0)        // Contraints for Hits
          Hits[x,y,0] = 0;                     // Initially, no cell has been hit...
   Hits[0,0,0] = 1;                            // ...but the origin

   forall (t in Pos, x,y in Coord: t > 0) // Chann. constr's for Hits
      ( (x = X[t] & y = Y[t]) => (Hits[x,y,t] = Hits[x,y,t-1] + 1) ) &
      ( (not (x = X[t] & y = Y[t])) => Hits[x,y,t] = Hits[x,y,t-1] );
   // Each cell is hit 0 or 1 times (string does not cross)
   forall (x,y in Coord, t in Pos) Hits[x,y,t] <= 1;
};
```

A.2 The Sailco inventory problem

```
int+ nbPeriods = ...;            range Periods 1..nbPeriods;
float+ demand[Periods] = ...;    float+ regularCost = ...;
float+ extraCost = ...;          float+ capacity = ...;
float+ inventory = ...;          float+ inventoryCost = ...;

var float+ regulBoat[Periods];   var float+ extraBoat[Periods];
var float+ inv[0..nbPeriods];

minimize ...                     // Objective function (omitted)
subject to {                     // Constraints
   inv[0] = inventory;
   forall(t in Periods) regulBoat[t] <= capacity;
   forall(t in Periods) regulBoat[t]+extraBoat[t]+inv[t-1] = inv[t]+demand[t];
};
```

A.3 The Blocks world problem

```
int nblocks = ...;
range Block 1..nblocks;               range BlockOrTable 0..nblocks;
BlockOrTable TABLE = 0;               range Time 1..2*nblocks;
range TimeWithZero 0..2*nblocks;      range bool 0..1;
BlockOrTable OnAtStart[Block] = ...;  BlockOrTable OnAtGoal[Block] = ...;

// MoveBlock[t] and MoveTo[t] refer to moves performed from time t-1 to time t
var Block MoveBlock[Time];
var BlockOrTable MoveTo[Time];
var BlockOrTable On[Block, TimeWithZero];      // Dependent guessed function
var bool Clear[BlockOrTable, TimeWithZero];    // Dependent guessed function
var TimeWithZero schLen;                       // Schedule length (to minimize)

minimize schLen
subject to {
   forall (b in Block) On[b,0] = OnAtStart[b]; // Initial state (time = 0);
   forall (b in Block, t in TimeWithZero) {    // Chann. constr's for Clear
      ( ( sum(b_up in Block) (On[b_up,t]=b) ) > 0 ) <=> (Clear[b,t] = 0); };
   forall (t in TimeWithZero) { Clear[TABLE,t] = 1; };

   forall (t in Time) {                        // Moves
     (MoveBlock[t] <> MoveTo[t]);
     (MoveTo[t] <> On[MoveBlock[t],t-1]);      // No useless moves
     (t <= schLen) => (
       (Clear[ MoveBlock[t], t-1 ] = 1) &      // Moving block must be clear
       (Clear[ MoveTo[t], t-1 ] = 1 )  &       // Target pos. must be clear
       (On[ MoveBlock[t], t ] = MoveTo[t]) );  // Chann. constr's for On
     forall (b in Block) {                     // Chann. constr's for On
         (t <= schLen) => (                    // (frame conditions)
            (b <> MoveBlock[t]) => (On[b,t] = On[b,t-1]) ) };
   };
   forall (b in Block) { On[b,schLen] = OnAtGoal[b]; }; // Final state
};
```

References

1. C. A. Brown, L. Finkelstein, and P. W. Purdom. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Proc. of 6th Intl. Conf. on Applied Algebra, Algebraic Algorithms and Error Correcting codes*, pages 99–110. Springer, 1988.
2. M. Cadoli and T. Mancini. Detecting and breaking symmetries on specifications. In *Proc. of the CP'03 Int. Workshop on Symmetry in CSPs*, pages 13–26, 2003.
3. M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. In *Proc. of KR'04*, pages 388–398. AAAI Press, 2004.

4. M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. In *Proc. of the ETAPS'04 CP+CV Int. Workshop*, pages 17–31, 2004.
5. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. In *Proc. of ESOP'01*, volume 2028 of *LNCS*, pages 387–401. Springer, 2001.
6. C. C. Chang and H. J. Keisler. *Model Theory, 3rd ed.* North-Holland, 1990.
7. B. M. W. Cheng, K. M. F. Choi, J. H.-M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.
8. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of KR'96*, pages 148–159, 1996.
9. P. Crescenzi, D. Goldman, C. H. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. *J. of Comp. Biology*, 5(3):423–466, 1998.
10. R. Dechter. Constraint networks (survey). In *Encyclopedia of Artificial Intelligence, 2nd edition*, pages 276–285. John Wiley & Sons, Inc., 1992.
11. H. D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1999.
12. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, Comparisons and Benchmarks. In *Proc. of KR'98*, pages 406–417, 1998.
13. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.
14. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
15. M. R. Garey and D. S. Johnson. *Computers and Intractability—A guide to NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.
16. E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. of AI*IA'99*, volume 1792 of *LNAI*, pages 84–94. Springer, 2000.
17. W. Hart and S. Istrail. HP Benchmarks. Available at `http://www.cs.sandia.gov/tech_reports/compbio/tortilla-hp-benchmarks.html`.
18. T. Hnich and T. Walsh. Why Channel? Multiple viewpoints for branching heuristics. In *Proc. of the CP'03 Int. Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation*, 2003.
19. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of AAAI'96*, pages 1194–1201, 1996.
20. K. F. Lau and K. A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *Macromolecules*, 22:3986–3997, 1989.
21. C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proc. of AAAI'00*. AAAI/MIT Press, 2000.
22. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
23. N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., 1980.
24. B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proc. of AAAI'00*, pages 182–187, 2000.
25. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
26. T. Walsh. Permutation Problems and Channelling Constraints. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of LPAR'01*, volume 2250 of *LNAI*, pages 377–391. Springer, 2001.
27. D. H. D. Warren. Extract from Kluzniak and Szapowicz APIC studies in data processing, no. 24, 1974. In *Readings in Planning*, pages 140–153. Morgan Kaufman, 1990.