Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 171 (2007) 985–1010
Artificial Intelligence
www.elsevier.com/locate/artint

Exploiting functional dependencies in declarative problem specifications ☆

Toni Mancini *, Marco Cadoli

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy

Received 4 June 2006; received in revised form 15 March 2007; accepted 30 April 2007

Available online 22 May 2007

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 modeling choices, e.g., to provide multiple viewpoints of the search space in order to increase constraint propagation. In either way, the recognition of dependencies greatly helps solvers, allowing them to avoid spending search on unfruitful branches, while maintaining the highest degree of declarativeness. By modeling constraint problem specifications as second-order formulae, we provide a characterization of functional dependencies in terms of semantic properties of first-order ones, and prove undecidability of the problem of their recognition. Despite such negative result, we advocate the (in many cases effective) possibility of using automated tools to mechanize this task. Additionally, we show how suitable search procedures can be automatically synthesized in order to exploit recognized dependencies. We present OPL examples of various problems, taken from bio-informatics, planning and resource allocation, and show how in many cases OPL greatly benefits from the addition of such search procedures. Moreover, we also give evidence that writing sophisticated ad-hoc search procedures that handle dependencies exploiting the peculiarities of the particular problem is a very difficult and error-prone task which in many cases does not seem to pay-off.

Keywords: Modeling; Reformulation; Second-order logic; Constraint satisfaction problems

1. Introduction

Declarative programming, and more specifically constraint programming, is becoming very attractive to solve different classes of problems, one of the main advantages of the approach being the fast prototyping and the high declarativeness exhibited by the problem models (also called "specifications"). Current systems for constraint solving (e.g., AMPL [19], OPL [43], DLV [31], SMODELS [36], and NP-SPEC [8]) allow the programmer to model her problem

☆ This paper is an extended and revised version of [M. Cadoli, T. Mancini, Exploiting functional dependencies in declarative problem specifications, in: Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA 2004), Lecture Notes in Artificial Intelligence, vol. 3229, Lisbon, Portugal, Springer, 2004, pp. 628–640].

* Corresponding author.
E-mail addresses: tmancini@dis.uniroma1.it (T. Mancini), cadoli@dis.uniroma1.it (M. Cadoli).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.04.017

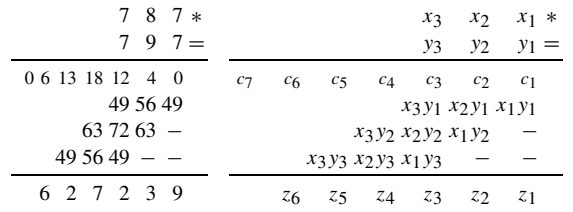


Fig. 1. Factoring instance 627239, $n = 6$, $b = 10$.

in a highly declarative way, supporting a neat separation of the specification from its instances. Such possibility allows the programmer to focus on structural and combinatorial aspects of the problem at hand before committing to actual input data, and hence permits problem modeling at a much higher level than that provided by the CSP framework.

However, it is well-known that the problem model obtained in this way is often not efficient, and much reasoning is required in order to reformulate it 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., [5,12]), the addition of implied constraints (cf., e.g., [41]), the deletion or abstraction of some of the constraints [3,16,20,23], and the use of redundant models, i.e., multiple viewpoints of the search space synchronized by channeling constraints, in order to increase constraint propagation [11,18,25,44]. However, many of these approaches either are designed for a specific constraint problem, or act at the instance level, and very little work has been done at the level of problem specification. Indeed, many of the properties of constraint problems amenable to optimizations strongly depend on the problem structure. Hence, their recognition naturally fits at the symbolic level of the specification, both from a methodological and an efficiency point of view.

Our research explicitly focuses on specification-level reasoning, with the goal of reformulating the declarative problem model submitted by the programmer into an equivalent one, more efficiently evaluable by solvers. In particular, in [6] we show how some of the constraints of a specification can be ignored in a first step, and then efficiently reinforced (i.e., without performing additional search, the so-called "safe delay" constraints), and provide a sufficient semantic criterion on the specification that can be used in order to recognize such constraints. Moreover, in [34] we tackle the issue of detecting structural (i.e., problem-dependent) symmetries, and breaking them by adding symmetry-breaking constraints to the problem specification.

In this paper we focus on another interesting property of constraint problems that is expected to benefit from reformulation, i.e., the functional dependencies that can hold among variables in declarative problem specifications. Informally, given a specification, a variable is said to be functional dependent on the others if, for every solution of every instance, its value is determined by those assigned to the others.

Functional dependencies are very common in problem specifications for different reasons: as an example, to allow the modeler 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. The following two examples show the use of dependent variables under the two afore-mentioned circumstances.

Example 1 *(Factoring [30,40]).* This problem is a simplified version of a well-known problem in public-key cryptography. Given a (large) positive integer Z, which is known to be the product of two different *prime* numbers (different from 1), it aims at finding its factors X and Y.

An intuitive formulation of factoring as a constraint problem, in order to deal with arbitrarily large numbers, amounts to encode the combinatorial circuit of integer multiplication. In particular, assuming the input integer Z having n digits (in base b) $z_1, \ldots, z_n$, we consider $2n$ variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ one for each digit (in base b) of the two factors, X and Y (with z_1, x_1, and y_1 being the least significant digits for Z, X, and Y, respectively). The domain for all these variables is $[0, b-1]$. In order to maintain information about the carries, $n+1$ additional variables $c_1, \ldots, c_{n+1}$ must be considered, with domain $[0, (b-1)^2 n/b]$.

As for the constraints (cf. Fig. 1 for the intuition, where $x_4, x_5, x_6, y_4, y_5, y_6$ are equal to 0, and are omitted for readability), they are the following:[1]

[1] Since integer Z is *assumed* to be the product of two prime numbers, constraints ensuring that X and Y are prime are not needed.

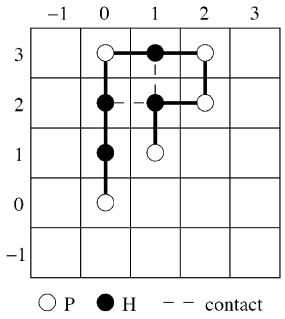


Fig. 2. HP 2D-Protein folding problem: A possible conformation for protein "PHHPHPPHP" with two contacts, and overall energy −2.

(1) Constraints on factors:
   (a) Factors must be different from 1, or, equivalently, $X \neq Z$ and $Y \neq Z$;
   (b) For every digit $i \in [1, n]$: $z_i = (c_i + \sum_{j,k \in [1,n]:\ j+k=i+1} x_j * y_k) \bmod b$;
(2) Constraints on carries:
   (a) Carry on the least significant digit is 0: $c_1 = 0$;
   (b) Carries on other digits: $\forall i \in [2, n+1]$, $c_i = (c_{i-1} + \sum_{j,k \in [1,n]:\ j+k=i} x_j * y_k)/b$ (integer division);
   (c) Carry on the most significant digit is 0: $c_{n+1} = 0$.

It is worth noting that, when a guess on the two factors X and Y (i.e., on variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$) has been made, values for variables $c_1, \ldots, c_{n+1}$ are completely determined, since they follow from the semantics of the multiplication. We denote such situation saying that variables $c_1, \ldots, c_{n+1}$ are functional dependent on $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$.

Example 2 *(HP 2D-Protein folding [29]).* This specification models a simplified version of a well-known problem 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 problem. The simplified version is known to be NP-complete [13] and very hard to solve in practice.

Given the sequence of amino-acids of the protein, i.e., a string over $\{H, P\}$ of length n, the problem aims to find a connected shape for it on a 2D grid (with coordinates in $[-(n-1), (n-1)]$, starting at a pre-determined position, e.g., $(0, 0)$), which is non-crossing, and such that the number of "contacts", i.e., the number of non-sequential pairs of H's for which the Euclidean distance of the positions is 1 is maximized (the overall energy is the opposite of the number of contacts). Fig. 2 shows a possible conformation of the protein "PHHPHPPHP", with overall energy −2.

Various alternatives for the search space exist: as an example, we can guess the position on the grid of each amino-acid, and then force the shape to be connected, non-crossing, and with minimal energy. However, a preferred approach that reduces the size of the search space (4^n points versus $(2n)^{2n}$) is to guess the shape of the protein as a connected path starting at $(0, 0)$, by guessing, for each index $i \in [1, n-1]$, the direction that the $(i+1)$-th amino-acid assumes with respect to the i-th one (directions can only be North, South, East, West).[2] The sequence of directions for the shape in Fig. 2 is: $[N, N, N, E, E, S, W, S]$.

However, choosing the latter model is not completely satisfactory. In fact, to express the non-crossing constraint, and 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.

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

Appendix B shows specifications for the above problems in the state-of-the-art declarative constraint modeling language OPL. However, they will be discussed in more detail in the forthcoming sections.

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

On the other hand, when using a declarative formalism such as that of CSPs, or declarative constraint modeling languages like, e.g., OPL, the programmer loses the power to distinguish among variables whose values have 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 programmer 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 variable 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.

The technique of avoiding branches on dependent variables has already been successfully applied for solving, e.g., SAT instances. As an example, it is shown in [22] 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 [32], 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 dependencies *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 give a formal characterization of functional dependencies on problem specifications* suitable to be checked by computer tools, and ultimately, *to transform the original specification by automatically suggesting an explicit search strategy* that exploits such dependencies, avoiding, or at least reducing, branches on dependent predicates. We experimentally show that such strategies, although general and simple, in many cases greatly enhance the performance of state-of-the-art constraint programming solvers like Ilog SOLVER,[3] and are often able to compete with procedures written ad-hoc for the given problem, that deeply exploit its structural peculiarities. This is good evidence that automated reasoning on problem specifications may be effective in order to improve system performance, while maintaining the highest level of declarativeness in the constraint model.

The outline of the paper is as follows: in Section 2 we introduce and justify the use of existential second-order logic as an abstract modeling language for problem specifications; then, in Section 3 we formally define functional dependencies in specifications, characterize them in terms of semantic properties of first-order formulae, and show that the problem of checking whether a dependence holds is undecidable. However, despite this negative result, we argue that current Automated Theorem Proving technology may be successfully used in order to mechanize the task of recognizing dependencies. Then, in Section 4 we show some examples from bio-informatics, planning and resource allocation, that exhibit dependencies, and present, in Appendix B, their formulations in the well-known constraint modeling language OPL. In Section 5 we describe our approach for exploiting detected dependencies, which consists in the automatic synthesis of a suitable search procedure that delays branches on dependent predicates, and experimentally show how, in all cases under evaluation, OPL benefits from this addition. Moreover, we show that the elaboration of ad-hoc, more complex search procedures, that strongly rely on the structure of the particular problem, although being a difficult and error-prone task in general, in many cases does not further improve performance. Finally, Section 7 is devoted to concluding discussions and description of future work.

2. Preliminaries

When dealing with problem specifications, the first choice to be made is the modeling language to be used. Current systems and languages for declarative constraint modeling, as those listed in Section 1, have their own syntax for

[3] Ilog SOLVER is the highly optimized backtracking-based Constraint Programming engine used by OPL.

describing constraint problems: AMPL, OPL, XPRESS^{MP}, ESRA and GAMS allow the representation of constraints by using algebraic expressions, while others, e.g., DLV, SMODELS, and NP-SPEC are rule-based languages, more specifically extensions of datalog. However, even if all such languages have a richer syntax and more complex constructs, from an abstract point of view all of them can be regarded as extensions of *existential second-order logic* (ESO) over finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the constraint modeling paradigm. In particular, in all of them it is possible to embed ESO queries, and the opposite encoding is also possible, as long as only finite domains are considered. Hence, as we show in this section, *ESO can be considered as the formal logical basis for virtually all available languages for constraint modeling*, being able to represent all search problems in the complexity class NP [17,38]. Intuitively, the relationship between ESO and available modeling languages is similar to that holding between assembler and high-level programming languages.

Moreover, since, as we will see in Section 3, the problem of detecting functional dependencies on ESO specifications reduces to checking semantic properties of first-order formulae, it is possible to use known results and techniques in order to mechanize such tasks. In particular, as advocated in related work [7] and briefly discussed at the end of forthcoming Section 3, this gives a promising new area of application for Automated Theorem Proving technology, which is undoubtedly one of the most important results achieved by Artificial Intelligence.

Existential second-order logic as a modeling language. Formally, an ESO specification describing a search problem π is a formula

$$\psi \doteq \exists \vec{\mathcal{S}}\, \phi(\vec{\mathcal{S}}, \vec{\mathcal{R}}), \tag{1}$$

where:

- The set of predicates $\vec{\mathcal{R}} = \{R_1^{ar(R_1)}, \ldots, R_k^{ar(R_k)}\}$ is the input relational schema, i.e., a fixed set of relations of given arities denoting the schema for all input instances of π. An instance $\vec{\mathcal{I}}$ of the problem is given, as it happens in current systems, as a *relational database* over the schema $\vec{\mathcal{R}}$, i.e., as an extension for all relations in $\vec{\mathcal{R}}$. All constants occurring in the database are *uninterpreted*, i.e., they don't have a specific meaning.
- Existentially quantified predicates in set $\vec{\mathcal{S}} = \{S_1^{ar(S_1)}, \ldots, S_n^{ar(S_n)}\}$ are called *guessed*, and their possible extensions, with tuples in the domain given by constants occurring in $\vec{\mathcal{I}}$ plus those occurring in ϕ, i.e., the so called Herbrand universe, encode points in the search space for problem π on instance $\vec{\mathcal{I}}$.
- ϕ is a closed first-order formula on the relational vocabulary $\vec{\mathcal{S}} \cup \vec{\mathcal{R}} \cup \{=\}$ ("=" is always interpreted as identity), that encodes the constraints an extension of predicates in $\vec{\mathcal{S}}$ must satisfy to be a solution of π on instance $\vec{\mathcal{I}}$.

An instance $\vec{\mathcal{I}}$ of π, encoded as a relational database having schema $\vec{\mathcal{R}}$, is *satisfiable* if and only if there exists a list of relations $\Sigma_1, \ldots, \Sigma_n$ (matching the list of guessed predicates $\vec{\mathcal{S}} = \{S_1, \ldots, S_n\}$) that, along with $\vec{\mathcal{I}}$, satisfies the first-order part of formula (1), i.e., such that

$$(\Sigma_1, \ldots, \Sigma_n, \vec{\mathcal{I}}) \models \phi.$$

In other words, the semantics of an ESO formula of the kind (1) is to find an extension of the guessed predicates that satisfies the constraints in ϕ for the given input instance $\vec{\mathcal{I}}$.

Since, by Fagin's theorem [17,38], a decision problem is in NP if and only if there exists an ESO formula that expresses it, such fragment of second-order logic can be definitively considered as an abstract modeling language for constraint problems.

An example should clarify the ESO formalism.

Example 3 *(Not-all-equal Sat [21, 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;
- $l^+(\cdot,\cdot)$; a tuple $\langle l, v\rangle$ is in l^+ iff l is the positive literal relative to the propositional variable v, i.e., v itself;

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

As an example, the CNF formula

$$(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$$

is a satisfiable instance of the problem, since the interpretation $\{x = true, y = false, z = true\}$ is a model of the formula that leaves at least one false literal in each clause.

Such an instance is encoded by the following set of relations $\vec{\mathcal{I}}$:

inclause

lit	*clause*
x_t	c_1
y_t	c_1
z_f	c_1
x_f	c_2
y_t	c_2
z_t	c_2

l^+

lit	*var*
x_t	x
y_t	y
z_t	z

l^-

lit	*var*
x_f	x
z_f	z

var

x
y
z

clause

c_1
c_2

where, to enhance readability, constants denoting positive and negative literals associated with variable v have been called v_t and v_f respectively.

An ESO specification for this problem is as follows:

$$\exists M^{(2)}\ \forall X, T\big(M(X,T) \rightarrow \big(var(X) \wedge (T = true \vee T = false)\big)\big) \wedge \tag{2}$$

$$\forall X\ \big(var(X) \rightarrow \exists T\ M(X,T)\big) \wedge \tag{3}$$

$$\forall X, T, T'\big(\big(M(X,T) \wedge M(X,T')\big) \rightarrow T = T'\big) \wedge \tag{4}$$

$$\forall C\ clause(C) \rightarrow \big[\exists L\ inclause(L,C) \wedge \forall V\ \big(l^+(L,V) \rightarrow M(V,true)\big) \wedge \big(l^-(L,V) \rightarrow M(V,false)\big)\big] \wedge \tag{5}$$

$$\forall C\ clause(C) \rightarrow \big[\exists L\ inclause(L,C) \wedge \forall V\ \big(l^+(L,V) \rightarrow M(V,false)\big) \wedge \big(l^-(L,V) \rightarrow M(V,true)\big)\big]. \tag{6}$$

The set of guessed predicates $\vec{\mathcal{S}}$ is a single binary predicate M which extensions are expected to encode possible assignments of truth values to the variables. To satisfy such a requirement, appropriate constraints (2–4) are needed. In particular, constraint (2) forces extensions of M to be sets of variable/truth-value pairs, while (3–4) constrain M to assign exactly one truth value to every variable. The need for constraints (2–4) directly follows from the semantics of ESO: declaring M as a binary guessed predicate implies that its extensions take values in the Herbrand domain, which is the set of constants occurring in the formula (in this example "*true*" and "*false*") plus those occurring in the problem instance, when one has been given. From the description given above such a set includes, besides symbols for variables, additional symbols for literals and clauses that must be forbidden in legal extensions of $M(\cdot,\cdot)$. Finally, as for the other constraints, (5) forces the assignment M to be a model of the formula, while (6) leaves in every clause at least one literal whose truth value is false.

The following extension for guessed predicate $M(\cdot,\cdot)$ is a solution of instance $\vec{\mathcal{I}}$ described above:

M

var	*value*
x	*true*
y	*false*
z	*true*

It is straightforward to see that such an extension for $M(\cdot,\cdot)$ satisfies all the constraints.

In order to facilitate the writing of specifications, additional built-in constructs are provided by current languages, in particular those aimed to represent typed relations, functions, e.g., arrays, bounded integers and arithmetics over them.

Such enrichments can actually be added also to the basic ESO framework. This eases expressions, and makes ESO specifications more compact and very close to their counterparts in state-of-the-art languages. In particular, syntactic sugar can be defined in order to let ESO be able to handle:

- typed guessed predicates,
- functions,
- bounded integers and arithmetics.

Support for typed guessed predicates and functions can be added in a straightforward way, since they can be realized by additional first-order constraints. A good example is given by the specification shown in Example 3, where constraints (2–4) force extensions of guessed predicate M to be *total functions* from the set of variables to the set $\{true, false\}$. As for numbers, instead, they can be simply handled by assuming that special relations in $\vec{\mathcal{R}}$ exist, that represent (pre-interpreted) bounded integers and operations among them. As an example, we can assume that a distinguished—finite—relation $number(\cdot) \in \vec{\mathcal{R}}$ exists, as well as predicates and operations needed to correctly express the problem specification (e.g., a binary relation $lt(x, y) \in \vec{\mathcal{R}}$ that contains tuples $\langle x, y\rangle$—with x, y being numbers—such that $x < y$, and a 3-ary relation $sum(\cdot, \cdot, \cdot) \in \vec{\mathcal{R}}$ containing tuples $\langle x, y, z\rangle$ such that $x + y = z$). Alternatively, in order to avoid pre-interpretation of constants and relations to handle numbers and arithmetics, we can guess additional relations that permit us to regard constants of the Herbrand domain $\mathcal{H}$ (or tuples of $\mathcal{H}^k$, for a suitable large k) as integers, by finding a total order over them. This technique requires us to guess additional predicates (cf., e.g., [33]).

We don't go into details of such extensions, but just observe that they *do not change the expressive power of ESO*.

In constraint problems, domains for variables are usually finite. Furthermore, for some of them, domains are "small", and independent of the particular instance. As an example, the set of possible directions that each amino-acid can assume with respect to the previous one in the Protein folding problem described in Example 2 is $\{N, S, E, W\}$ independent of the instance. The same happens for the truth values that can be assigned to variables in the Not-all-equal SAT problem of Example 3, which are $\{true, false\}$. In such cases, there exist alternative, but equivalent, ESO specifications.

In general, given an n-ary guessed predicate P with one of the arguments over a domain of size m (independent of the instance), we can always replace it by m $(n-1)$-ary guessed predicates $P_1, \ldots, P_m$, one for each such value. The remainder of the specification must be rewritten accordingly. Such a process is called *unfolding* (of P according to the given argument). As an example, in the following we give an unfolded ESO specification for the Not-all-equal SAT problem.

Example 4 *(Not-all-equal SAT unfolded (Example 3 continued)).* This specification has been obtained from that of Example 3 by unfolding the guessed predicate $M(\cdot,\cdot)$ according to the second argument (having domain $\{true, false\}$), hence obtaining 2 monadic guessed predicates, that we call T and F (instead of M_{true} and M_{false}) to enhance readability. Extensions of such predicates are forced to contain the set of variables assigned to true and false, respectively.

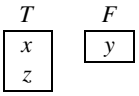
$$\exists T\, F\ \ \forall X\ var(X) \leftrightarrow T(X) \vee F(X) \wedge \tag{7}$$

$$\forall X\ \neg\big(T(X) \wedge F(X)\big) \wedge \tag{8}$$

$$\forall C\ clause(C) \rightarrow \big[\exists L\ inclause(L, C) \wedge \forall V\ \big(l^+(L, V) \rightarrow T(V)\big) \wedge \big(l^-(L, V) \rightarrow F(V)\big)\big] \wedge \tag{9}$$

$$\forall C\ clause(C) \rightarrow \big[\exists L\ inclause(L, C) \wedge \forall V\ \big(l^+(L, V) \rightarrow F(V)\big) \wedge \big(l^-(L, V) \rightarrow T(V)\big)\big]. \tag{10}$$

The following extensions for guessed predicates $T(\cdot)$ and $F(\cdot)$ encode the same solution of instance $\vec{\mathcal{I}}$ described in Example 3:



ESO vs. CSP. As it has been shown above, an ESO formula may be used to model the structure of a decision problem (specification), independently of the input data (instance). As already discussed in Section 1, current systems for constraint programming that support declarative modeling clearly separate the problem specification from its instances, offering specification languages which are essentially ESO (cf. Appendix B for a short description of the OPL language).

However, when input data is given (by fixing an extension for predicate symbols in set $\vec{\mathcal{R}}$), a classical Constraint Satisfaction Problem (CSP, in the sense of [14]) is obtained, i.e., a triple $\langle \vec{\mathcal{X}}, \vec{\mathcal{D}}, \vec{\mathcal{C}} \rangle$ where:

- $\vec{\mathcal{X}}$ is a linearly ordered set of variables,
- $\vec{\mathcal{D}}$ is a linearly ordered set of domain values, one for each variable, and
- $\vec{\mathcal{C}}$ is a set of constraints, each one defined on a linearly ordered subset of the variables (the *constraint scope*) and encoded as a subset of the Cartesian product of the respective domains (the *constraint relation*).

This process is usually called *grounding*. Of course, there are in general several ways to ground a specification, since there may be different ways to choose variables and values, as the following example shows.

Example 5 *(Not-all-equal SAT as CSP).* Let us consider the specification for the Not-all-equal SAT problem shown in Example 3, together with instance $\vec{\mathcal{I}}$. A straightforward grounding of the specification on instance $\vec{\mathcal{I}}$ is as follows:

- $\vec{\mathcal{X}} = \{x, y, z\}$, i.e., one CSP variable for each variable of the CNF;
- $\vec{\mathcal{D}} = \{D_x, D_y, D_z\}$, with $D_x = D_y = D_z = \{true, false\}$;
- As for the set of constraints $\vec{\mathcal{C}}$, we have the following ones:
  - For each clause c_i ($i \in \{1, 2\}$) we have a constraint $C_i^{(5)}$ defined on the variables that occur in c_i. Such constraints together encode the one denoted by (5) in the specification. Hence, $C_1^{(5)}$ is defined over $\{x, y, z\}$ and its relation will contain those 3-tuples with components in $\{true, false\}$ that encode assignments to the variables that satisfy clause c_1, i.e., $C_1^{(5)} = \{true, false\}^3 - \{\langle false, false, true\rangle\}$. Analogously, $C_2^{(5)}(x, y, z) = \{true, false\}^3 - \{\langle true, false, false\rangle\}$.
  - Similarly, for each clause c_i ($i \in \{1, 2\}$) we have a constraint $C_i^{(6)}$ defined on the variables that occur in c_i. Such constraints together encode the one denoted by (6) in the specification. In particular, $C_1^{(6)}$ and $C_2^{(6)}$ will contain those 3-tuples with components in $\{true, false\}$ that encode assignments to the variables for which at least one literal in clauses c_1 and c_2, respectively, is false, i.e., $C_1^{(6)}(x, y, z) = \{true, false\}^3 - \{\langle true, true, false\rangle\}$ and $C_2^{(6)}(x, y, z) = \{true, false\}^3 - \{\langle false, true, true\rangle\}$.

Of course, optimizations are possible to this encoding: as an example, constraints $C_i^{(5)}$ and $C_i^{(6)}$ can be merged together, for each $i \in \{1, 2\}$.

Finally, we observe that the presented CSP encoding is correct because constraints (3–4) of the ESO specification force $M(\cdot, \cdot)$ to be a total *function* (i.e., a total *mono-valued* relation) from the set of variables of the CNF to $\{true, false\}$. If it were not the case, we would e.g. define a CSP *boolean* variable for each CNF-variable/truth-value pair, to account for all possibilities.

3. Definitions and formal results

In Section 2 we presented ESO, explained how it can be regarded as an abstract modeling language for problem specifications, and how it relates to the CSP framework. In this section we formally discuss what we mean by functional dependencies in an ESO specification, provide a logical characterization for them, and show how the problem of checking whether a dependence holds reduces to verifying semantic properties of first-order formulae.

In Section 1 we gave instances of functional dependencies in constraint problem specifications. As an example, in the Factoring problem (cf. Example 1), when a guess is made on the two factors X and Y, variables for carries are determined by the semantics of integer multiplication. Similarly, in Protein folding (cf. Example 2), absolute positions of the amino-acids are determined once we guess the protein shape.

To start with a simpler example, let us consider again the Not-all-equal SAT problem in the unfolded version of Example 4.

Example 6 *(Not-all-equal Sat, Example 4 continued).* Guessed predicate F is dependent on T because, once a guess has been made on the extension of the latter, there exists at most a single extension of the former leading to a solution. This is implied by constraints (7–8), from where it logically follows that:

$$\forall X \quad F(X) \leftrightarrow var(X) \wedge \neg T(X), \tag{11}$$

i.e., that CNF variables assigned to *false* are exactly those which are not assigned to *true*. The reverse dependence obviously also hold, hence each guessed predicate is dependent on the other (we say that they are *mutually* dependent).

Example 6 shows functional dependencies *among distinct guessed predicates*. However, if we consider the first specification of this problem given in Example 3, the same dependencies hold *among tuples of the same guessed predicate* $M(\cdot,\cdot)$. However, by unfolding specifications in which guessed predicates exhibit dependencies among their tuples, we can always reduce ourselves to the former case. For this reason, in what follows we restrict ourselves to dependencies among distinct guessed predicates.[4]

To simplify notation, given a list of predicates $\vec{T}$, we write $\vec{T}'$ to represent 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, $\vec{T} \equiv \vec{T}'$ will be a shorthand for the formula

$$\bigwedge_{T \in \vec{T}} \forall \vec{X} \quad T(\vec{X}) \leftrightarrow T'(\vec{X}),$$

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

The following definition formally characterizes functional dependencies among guessed predicates in an ESO specification.

Definition 1 *(Functional dependence of a set of predicates in a specification).* Given a problem specification on input schema $\vec{\mathcal{R}}$

$$\psi \doteq \exists \vec{\mathcal{S}} \vec{\mathcal{P}}\, \phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}),$$

in which the set of guessed predicates is partitioned into $\vec{\mathcal{S}}$ and $\vec{\mathcal{P}}$, predicates in $\vec{\mathcal{P}}$ *functionally depend* on those in $\vec{\mathcal{S}}$ if, for each instance $\vec{\mathcal{I}}$ of $\vec{\mathcal{R}}$ and for each pair of interpretations $\langle \vec{\Sigma}, \vec{\Pi} \rangle$, $\langle \vec{\Sigma}', \vec{\Pi}' \rangle$ of $(\vec{\mathcal{S}}, \vec{\mathcal{P}})$ it holds that, if

(1) $\langle \vec{\Sigma}, \vec{\Pi} \rangle \not\equiv \langle \vec{\Sigma}', \vec{\Pi}' \rangle$, and
(2) $\vec{\Sigma}, \vec{\Pi}, \vec{\mathcal{I}} \models \phi$, and
(3) $\vec{\Sigma}', \vec{\Pi}', \vec{\mathcal{I}} \models \phi$,

then $\vec{\Sigma} \not\equiv \vec{\Sigma}'$.

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

[4] We observe that unfolding is used here just as a formal step to let the specification suit the formal framework we are going to define, and does not need to be performed in practice. Hence, we can handle also those cases where the number of guessed predicates obtained by unfolding is instance dependent.

It is worth noting that Definition 1 is strictly related to the concept of *Beth implicit definability*, well-known in logic, which is as follows: given a (first-order) logical formula $\phi(\vec{\mathcal{P}})$ over a set of predicates $\vec{\mathcal{P}}$, ϕ is said to implicitly define predicate $P \in \vec{\mathcal{P}}$ if every $(\vec{\mathcal{P}} - \{P\})$-structure has at most one expansion to a $\vec{\mathcal{P}}$-structure satisfying ϕ (cf., e.g., [10]). We will further discuss this relationship in Section 5.

In what follows, 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 checking semantic properties of a first-order formula (proofs are in Appendix A).

Theorem 1. *Let $\psi \doteq \exists\vec{\mathcal{S}}\vec{\mathcal{P}}\ \phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$ be a problem specification with input schema $\vec{\mathcal{R}}$. Guessed predicates in set $\vec{\mathcal{P}}$ functionally depend on those in $\vec{\mathcal{S}}$ if and only if the following first-order formula is a tautology*:

$$\left[\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}) \wedge \phi(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}) \wedge \vec{\mathcal{S}}\vec{\mathcal{P}} \not\equiv \vec{\mathcal{S}}'\vec{\mathcal{P}}'\right] \rightarrow \vec{\mathcal{S}} \not\equiv \vec{\mathcal{S}}'. \tag{12}$$

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

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

This undecidability result shows that it is not always possible to mechanize the task of establishing whether a given dependence holds. This is a major problem along the way of providing automated techniques to perform symbolic problem reformulation in order to optimize the declarative specifications given by the user into ones more efficiently solvable.

However, despite this negative theoretical result, related work [7] shows that, in many practical circumstances, the task of detecting functional dependencies can be effectively and often efficiently performed by automated tools. We don't go here into details, but just give, in Fig. 3, an example of how Theorem 1 can be exploited in order to let a first-order theorem prover (namely OTTER [35]) check whether predicate F is functional dependent on T in the Not-all-equal SAT specification of Example 4. The OTTER encoding of formula (12) strictly follows its structure, and thus can be easily derived automatically, even from a specification given in an implemented modeling language such as OPL, which, despite some syntactic sugar, is actually very similar to ESO (cf. Appendix B). In particular, parts 1 and 2 of the encoding given in Fig. 3 define $\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$ and $\phi(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}})$ coherently with constraints of the ESO specification (we used auxiliary propositional variables for single constraints, as explained in [7]), while parts 3 and 4 encode "$\vec{\mathcal{S}}\vec{\mathcal{P}} \equiv \vec{\mathcal{S}}'\vec{\mathcal{P}}'$" and "$\vec{\mathcal{S}} \equiv \vec{\mathcal{S}}'$" respectively. Finally, part 5 checks whether the negation of formula (12) is a contradiction.

OTTER was able to prove that the encoded formula (negation of (12)) is a contradiction, and hence the existence of the functional dependence, in a few hundreds of seconds. We address the reader to [7] for an exhaustive discussion and an experimental evaluation on several problems that highlights how such tools may be effective in practice to perform the task of checking (among other structural properties of constraint specifications) functional dependencies.

4. Further examples

In this section we present some problem specifications which exhibit functional dependencies among their guessed predicates. Since they have several elaborated constraints, we don't give their formulations as ESO formulae, but show, in Appendix B, their specifications in the well known language for constraint modeling OPL.

Example 7 *(Factoring, Example 1 continued).* An OPL specification for this problem is presented in Appendix B.1. The input schema and the set of guessed predicates defining the search space are as follows:

Input schema: array of integers `Z[]` denoting the input integer (with the least indices denoting the least significant digits), and integer variables `base` and `digitsZ` denoting its base and number of digits, respectively.

Guessed predicates: arrays `X[]` and `Y[]` of digits in base `base` denoting the two factors, and an array of carries `carry[]`. Further guessed variables are needed for technical reasons, cf. specification in Appendix B.1.

```
set(auto).
formula_list(usable).

% 1. Encoding of phi(T,F,...)
  cov  <-> (all x (var(x) <-> (T(x) | F(x)))).
  disj <-> (all x (-(F(x) & T(x)))).
  sat  <-> (all c exists l all v (clause(c) ->
             (inclause(l,c) &
              (litP(l,v) -> T(v)) & (litN(l,v) -> F(v))))).
  nae <-> (all c exists l all v (clause(c) ->
            (inclause(l,c) &
              (litP(l,v) -> F(v)) & (litN(l,v) -> T(v))))).
  phi <-> cov & disj & sat & nae.

% 2. Encoding of phi_prime(T_prime,F_prime,...)
  cov_prime  <-> (all x (var(x) <-> (T_prime(x) | F_prime(x)))).
  disj_prime <-> (all x (-(F_prime(x) & T_prime(x)))).
  sat_prime  <-> (all c exists l all v (clause(c) ->
            (inclause(l,c) &
             (litP(l,v) -> T_prime(v)) & (litN(l,v) -> F_prime(v))))).
  nae_prime <-> (all c exists l all v (clause(c) ->
            (inclause(l,c) &
             (litP(l,v) -> F_prime(v)) & (litN(l,v) -> T_prime(v))))).
  phi_prime <-> cov_prime & disj_prime & sat_prime & nae_prime.

% 3. Encoding of (T,F) <-> (T_prime, F_prime)
  equivTF <-> (all x ((T(x) <-> T_prime(x)) & (F(x) <-> F_prime(x)))).

% 4. Encoding of T <-> T_prime
  equivT <-> (all x ((T(x) <-> T_prime(x)))).

% 5. It is not true that "F is dependent on T"
  -( (phi & phi_prime & -equivTF) -> -equivT ).

end_of_list.
```

Fig. 3. OTTER input file that checks whether predicate F is dependent on T in the Not-all-equal problem specification of Example 4.

As for the constraints, we have OPL encodings of those described in Example 1. As already observed the array `carry[]` is functionally dependent on `X[]` and `Y[]`.

Example 8 *(The HP 2D-Protein folding problem, Example 2 continued).* As already stated in Example 2, rather than guessing the position on the grid of each amino-acid in the sequence, we chose to represent the shape of the protein giving, 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 $(0, 0)$). However, in order to express the non-crossing constraint, and to compute the value of the objective function, absolute coordinates of each amino-acid in the sequence must be calculated and maintained.

An OPL specification for this problem is shown in Appendix B.2. The input schema and the set of guessed predicates are given as follows:

Input schema: Array `seq[]` of amino-acids (in the set {H,P}), and integer variable `n` encoding the string length.
Guessed predicates: Array `Moves[]` with `n-1` components in the set {`N`, `S`, `E`, `W`} encoding the moves of the "string head", plus integer arrays `X[]` and `Y[]` of length `n` representing the absolute positions of each amino-acid.

As for the constraints, we force the protein shape to start at (0,0) and to be non-crossing: such constraints are expressed in terms of `X[]` and `Y[]` variables, that are linked to those of `Moves[]` by appropriate channeling constraints. As already observed, guessed predicates `X[]` and `Y[]` are functionally dependent on `Moves[]`. Finally, the objective function maximizes the number of non-sequential pairs of H amino-acids for which the Euclidean distance of the positions is 1.

A note on the non-crossing constraint is in order. The most natural formulation of this constraint uses $\mathcal{O}(n^2)$ binary inequalities, i.e., forbidding the existence of two different elements t and t' such that `X[t] = X[t']` and `Y[t] = Y[t']`. However, we considered also a second formulation for HP 2D-Protein folding, where the non-crossing constraint is modeled differently. In particular, a new guessed predicate `Hits[]` is used, in order to maintain, for every position on the grid, the number of amino-acids of the protein that are placed on it at each point during the

construction of the shape. For each position this number cannot be greater than 1, which implies that the string does not cross. In this specification, whose OPL code is presented in Appendix B.2, again the guessed predicate `Hits[]` is dependent on `Moves[]`.

Example 9 *(The Sailco inventory problem [43, Section 9.4]).* 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).

Appendix B.3 shows an OPL model for this problem. In particular, the instance schema and the set of guessed predicates are as follows:

Instance schema: The number of periods `nbPeriod`, integer array `demand[]` stating the demand of each period, plus integers `regularCost`, `extraCost`, `inventoryCost`, `capacity` and `inventory`.

Guessed predicates: Arrays `regulBoats[]` and `extraBoats[]` that guess, for each period, the number of regular and extra boats to be produced, plus array `inv[]` maintaining the number of boats stored in the inventory in each period.

As for the constraints, they force the inventory to contain `inventory` boats initially, impose that, at any period, a maximum of `capacity` regular boats can be produced, and define the number of boats stored in the inventory at each period. Finally, the objective function minimizes the overall production cost.

From the specification it can be observed that the number of boats stored at period `t` > 0 (i.e., `inv[t]`) is defined in terms of the number of regular and extra boats produced in period `t` by the following relationship: `inv[t] = regulBoats[t] + extraBoats[t] - demand[t] + inv[t-1]`. Such a relationship makes guessed predicate `inv[]` functionally dependent on `regulBoats[]` and `extraBoats[]`.

Example 10 *(The Blocks world problem [37,45]).* 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 on another block. Given the initial and the desired configurations of the blocks, the problem amounts to find a minimal sequence of moves that achieves the desired configuration starting from the initial one. 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 onto 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 onto the table before building the desired configuration.

In our formulation, given in Appendix B.4, the instance schema and the set of guessed predicates are defined as follows:

Instance schema: input is given as an integer `nblocks`, i.e., the number of blocks, and arrays `OnAtStart[]` and `OnAtGoal[]`, encoding, respectively, the initial and desired configurations.

Guessed predicates: Arrays `MoveBlock[]` and `MoveTo[]` that 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 have arrays `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.

As for the constraints, we state that, at any time point, only one move can be performed, that selects a clear block and puts it onto the table or onto another clear block (with the rest of the configuration remaining identical, i.e., the so-called "frame axiom"). Moreover, we impose that the starting and final configurations are those defined by `OnAtStart[]` and `OnAtGoal[]`. Finally, channeling constraints that define guessed predicates `On[]` and `Clear[]` are given. The latter constraints give evidence that guessed arrays `On[]` and `Clear[]` are functionally dependent on `MoveBlock[]` and `MoveTo[]`.

Finally, the objective function minimizes the number of moves needed to reach the final configuration.

5. Exploiting functional dependencies

In previous sections we discussed on how often functional dependencies arise in declarative constraint problem specifications and how negatively they may affect solvers' performance. By using ESO as a modeling language, we formally characterized dependencies in terms of first-order logic, leading to the possibility of making use of automated tools in order to mechanize the task of their recognition. Despite the undecidability of the general problem (cf. Theorem 2), in related work [7] we experimentally show that this approach is feasible in many practical cases, thus suggesting a brand new and promising application area and new challenges for Automated Theorem Proving technology.

Now, being able to recognize, given a problem specification, whether a subset $\vec{\mathcal{P}}$ of the guessed predicates is functionally dependent on the others, the opportunity of exploiting such a dependence arises, with the ultimate goal of improving solver's performance. Different approaches can be adopted in principle, all of them aiming at *excluding* from the search space predicates in $\vec{\mathcal{P}}$. In this section, we comment on some of them, and present a simple and general technique that may be successfully applied by constraint programming systems in order to automate even this second task.

The most natural technique to handle a dependent predicate $P \in \vec{\mathcal{P}}$ is, arguably, to *substitute* all occurrences of P occurring in the specification with its *definition*, i.e., with the formula that defines it in terms of the others (cf., e.g., formula (11) in Example 6). However, this approach is unfeasible in general. In fact, 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, [10]). In particular, given a problem specification $\psi \doteq \exists \vec{\mathcal{S}} \vec{\mathcal{P}}\, \phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$, guessed predicates in set $\vec{\mathcal{P}}$ functionally depend on those in $\vec{\mathcal{S}}$ if and only if the first-order formula $\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$ implicitly defines predicates in $\vec{\mathcal{P}}$, i.e., if every $\langle \vec{\mathcal{S}}, \vec{\mathcal{R}} \rangle$-structure has at most one expansion to a $\langle \vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}} \rangle$-structure satisfying $\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$.

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., [15], and the intrinsically inductive definition of guessed function `inv[]` in Example 9). On the other hand, a second-order explicit definition of a dependent predicate would not be adequate, since new quantified predicates would be added to the specification, and, moreover, the obtained specification may not be in ESO any more.

Although this approach is not feasible in general, in some cases first-order formulae that explicitly define dependent predicates in terms of the others indeed exist (cf., e.g., Example 6). However, even in such cases, replacing all occurrences of predicates in $\vec{\mathcal{P}}$ with such formulae is likely to lead to longer and more complex constraints, and thus to worsen performance.

The second general approach to deal with dependencies (that we will exploit in this paper) is that of *instructing* the search engine to not consider dependent predicates as part of the search space, and, instead, to *compute*, given an extension of predicates in $\vec{\mathcal{S}}$, the corresponding extension of predicates in $\vec{\mathcal{P}}$. To this end, we assume a language that allows us to define an explicit search strategy. In particular, since its description may depend on the particular language, in the remaining of the paper we use the syntax of the constraint language OPL.

OPL does not require a search strategy to be provided by the programmer, since it automatically uses default strategies (based on highly optimized versions of dynamic value ordering heuristics, cf., e.g., [1,28,39]) 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 means of an optional part of the problem model called "search procedure". Appendix B shows a brief description of what a specification in OPL looks like (and its similarities with ESO), as well as specifications for the examples discussed in Section 4.

The approach of adding a search procedure to the problem model in order to exploit dependencies is very powerful, but can be very costly in terms of human effort, since the algorithm needed to compute the extension of dependent predicates corresponding to one of those in $\vec{\mathcal{S}}$ strongly depends on the peculiarities of the given problem, and hence needs to be designed and implemented by the modeler, with the consequence of a great lowering of the declarativeness of the process. Nonetheless, this approach is likely to be the most efficient one, and is often used in practice, even if

it is intrinsically error-prone (since procedural aspects have to be taken into account), and can easily lead to unsound procedures (that, e.g., terminate before exploring the whole problem search space).

Example 11 *(The HP 2D-Protein folding problem).* One of the most intuitive search procedures that can be added to the specification for this problem in order to exploit the dependence of guessed predicates `X[]` and `Y[]` on `Moves[]` is the one that labels variables in `Moves[]` in ascending order (i.e., `Moves[0]` first, then `Moves[1]`, etc.) and, after having guessed `Moves[t]`, *computes* `X[t+1]` and `Y[t+1]`, starting from `X[t]`, `Y[t]` (with `X[0]` = `Y[0]` = 0) and the last move, according with the following rules:

- If `Moves[t] = N`, then `X[t+1] = X[t]` and `Y[t+1] = Y[t]+1`;
- If `Moves[t] = S`, then `X[t+1] = X[t]` and `Y[t+1] = Y[t]-1`;
- If `Moves[t] = E`, then `X[t+1] = X[t]+1` and `Y[t+1] = Y[t]`;
- If `Moves[t] = W`, then `X[t+1] = X[t]-1` and `Y[t+1] = Y[t]`.

An OPL formulation of this search procedure is given in Appendix B.2.

Of course, when adding such a search procedure, channeling constraints in the specification that define guessed variables `X[]` and `Y[]` in terms of `Moves[]` could be safely removed: the synchronization between the two viewpoints of the search space is guaranteed by the search procedure itself. However, this practice is very dangerous from a methodological standpoint, since a strong coupling would be introduced between the declarative and procedural parts of the problem model. This is because such constraints are conceptually part of the problem specification (without them, the specification would be incorrect), and if the programmer, later on, chooses to change the search procedure, then she must add the channeling constraints back, or handle the synchronization issue in some other way.

On the other hand, it should be clear that leaving such constraints in the specification does not introduce appreciable costs during the solving process, since the behavior of the search procedure guarantees that they are always satisfied.

It is immediate to observe that this search procedure is very effective, since it *maximally reduces* the size of the search space, actually *excluding* `X[]` and `Y[]` from being part of it, and guesses directions of the protein "head" from the first to the last amino-acid. However, such a procedure is very unlikely to be the output of a mechanized task, once the functional dependence of `X[]` and `Y[]` over `Moves[]` has been detected, since it relies on a deep analysis of the problem model, that has to be provided by the programmer.

On the other hand, our goal is to automate the synthesis of suitable search procedures that exploit functional dependencies, even if less effective than those that can be written by the modeler. To this end, simple and general schemas must be followed, that do not rely on structural peculiarities of the given problem.

The idea that we are going to show is much easier to automate. It aims to *enforce a preference order on the variables to branch on*, in such a way that those corresponding to dependent predicates are delayed as long as possible. As an example, in the HP 2D-Protein folding problem, the algorithm may *first* branch on variables of the `Moves[]` array, and *then* on those of `X[]` and `Y[]`. In this way, although dependent predicates are not excluded from the search space, constraint propagation will typically effectively (and efficiently) reduce the active domains associated to dependent variables, after a guess on defining ones has been performed, especially when tight channeling constraints are available in the problem model (as in this case).

Such a technique can be implemented in the OPL language in a very simple way, by using the high-level `generate()` construct, which receives a guessed predicate as input and forces the algorithm to generate all possible extensions for it, leaving the policy for the generation (i.e., the ordering of variables to branch on and of values to be assigned to them) to the system defaults. Of course, multiple occurrences of `generate()`, with different guessed predicates as arguments, are allowed.

Hence, given a problem specification in which a set $\vec{\mathcal{P}} = \{\mathtt{P1}, \ldots, \mathtt{Pn}\}$ of the guessed predicates is functionally dependent on the others (set $\vec{\mathcal{S}} = \{\mathtt{S1}, \ldots, \mathtt{Sm}\}$), a search procedure that forces OPL to first branch on predicates in $\vec{\mathcal{S}}$ is the following:

```
// DDP - Delay branches on Dependent Predicates
search {
   generate(S1);
```

```
    ...
    generate(Sm);

    generate(P1);
    ...
    generate(Pn);
};
```

which additionally leaves the policies of the generation of extensions for the predicates in $\vec{\mathcal{S}}$ and $\vec{\mathcal{P}}$ to the system defaults (actually, since the OPL `generate()` construct accepts only a single predicate as input, as a side effect, we fix preference orders among {`S1`, ..., `Sn`} and {`P1`, ..., `Pm`}; the orders chosen may, of course, affect performance).

In the following, we refer to this schema as *DDP* ("Delay branches on Dependent Predicates"). Such a schema is simple, general, and it is very easy to automate. Interestingly, as shown in Section 6, it also has very good performance, being able to compete, in many cases, with more complex approaches, like that shown in Example 11.

Some observations on the structure of the DDP schema are in order: it can be wondered whether, in many cases (e.g., when appropriate channeling constraints exist in the specification), constraint propagation is sufficient to isolate, once a guess on predicates in $\vec{\mathcal{S}}$ has been made, the unique correct extension for predicates in $\vec{\mathcal{P}}$. This is the case, e.g., for Protein folding (cf. its OPL specification in Appendix B.2). In those circumstances, `generate(P1);...; generate(Pn);` could be safely removed from the search procedure. However, it must be observed that, by the definition of functional dependence (cf. Definition 1), it follows that the problem of isolating the only correct extension for dependent predicates once a guess on the defining ones has been made is, in general, a (sub-)problem in NP which is guaranteed to have exactly one solution. Since such problems are believed to be as complex as arbitrary NP problems (cf., e.g., [9,38,42]), constraint propagation alone (a polynomial-time algorithm) may not suffice. On the other hand, it is worth observing that, in those cases where constraint propagation is sufficient to isolate the only correct value for dependent predicates, "generating" extensions for them does not add any considerable overhead to search, since the sets of their possible extensions (i.e., the active domains of the corresponding CSP variables obtained after grounding) has been already reduced by propagation to singletons; this step would thus be made in constant time.

Finally, for some specifications, sets $\vec{\mathcal{S}}$ and $\vec{\mathcal{P}}$ are *interchangeable*. This intuitively happens when the modeler adopts multiple viewpoints of the search space (cf., e.g., Example 8, 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., $\vec{\mathcal{S}}$), may involve the size of the associated search space, but other and smarter approaches can be used, like the amenability of the various constraints to propagation (cf., e.g., [25]).

6. Experiments

To test the effectiveness of adding the DDP search procedure presented in Section 5, we experimented with OPL on many of the problems described above:[5]

- HP 2D-Protein folding (cf. Example 8) on benchmark instances, some of them taken from [24];
- Blocks world (cf. Example 10) on structured instances, used as benchmarks in [26];
- Factoring (cf. Example 1) on instances denoted by numbers of 13/14 digits.

For each of them, we solved all instances without any search procedure (hence, relying on the OPL default strategy) and with the general DDP search procedure suggested in Section 5 that instructs the search engine to first branch on defining variables, and then on dependent ones. The first result of our experiments is that *in all cases, adding the DDP search procedure significantly speed-ups the computation*. This is evidence that reasoning on the problem model is a promising approach to boost performance.

[5] All specifications and instances are available at http://www.dis.uniroma1.it/~ tmancini/index.php?currItem=research.publications.webappendices.mancini-cadolil06|dependencies.

Table 1
OPL solving times for benchmark instances of the HP 2D-Protein folding problem (model with binary inequalities for the non-crossing constraint), with and without search procedures

Length	max. contacts	No search proc. (default strategy)	With search procedure					
			DDP	Saving %	1.	Saving %	2.	Saving %
14	2	33.85	33.40	1.33	38.13	−12.64	35.55	−5.02
14	5	45.02	39.23	12.86	45.74	−1.60	41.64	7.51
16	6	124.16	114.03	8.16	130.73	−5.29	117.90	5.04
16	7	24.13	22.17	8.12	25.35	−5.06	22.91	5.06
17	6	323.66	288.22	10.95	326.61	−0.91	300.33	7.21
17	6	2818.51	2149.71	23.73	2401.81	14.78	2220.18	21.23
18	4	576.85	487.68	15.46	552.50	4.22	509.59	11.66
Total time		3946.18	3134.44	20.57	3520.87	10.78	3248.10	17.69

The second important result of the experiments is quite surprising: *in almost all cases, a more sophisticated search procedure, ad-hoc written by the programmer with significant effort, great lowering of declarativeness, and at the risk of being unsound, does not further improve performance, and sometimes performs even worse than DDP*.

All the experiments used Ilog SOLVER v. 5.3, invoked through OPLSTUDIO 3.61, on a 2 CPU Intel Xeon 2.4 GHz computer, with 2.5 GB RAM and Linux v. 2.4.18-64GB-SMP.

Results are shown in Tables 1–4 for HP 2D-Protein folding, Factoring, and Blocks world, and are briefly commented on in what follows.

HP 2D-Protein folding We made experiments with the specification shown in Appendix B.2, that uses binary inequalities to model the non-crossing constraint.

As for the search procedures added in order to exploit the dependence of `X[]` and `Y[]` on `Moves[]`, we used the general DDP schema, and the following two additional ones, ad-hoc built relying on the structural peculiarities of the problem:

(1) The procedure described in Example 11;
(2) A simplification of the above one, that, for each `t` in increasing order, after having guessed `Moves[t]`, does not explicitly compute values for `X[t+1]` and `Y[t+1]`, but leaves the engine free to search for suitable values using the channeling constraints and OPL defaults for the search strategy.

It is interesting to observe that such search procedures, although exploiting the structure of the particular problem model much better than DDP (of course at the cost of additional programming effort) have *worse performance*. Such behavior is common, with few interesting exceptions, to several problems, and is analyzed at the end of the section.

As already mentioned in Example 8, we wrote a second formulation for this problem, in which the non-crossing constraint is defined in terms of the additional ternary guessed predicate `Hits[]`. In this case, again this predicate is dependent on `Moves[]`. However, this alternative formulation turned out to be of very low quality, being much less efficient than the previous one. Interestingly, in this case, benefits of adding the DDP search procedure are very impressive. Detailed results are given in Table 2. This is a good example of how a reasoning task on a problem specification is able to recover some inaccuracies made by designers, when writing bad models.

Blocks world. Also results for the Blocks world problem show that delaying branches on dependent predicates greatly speed-ups the computation. Besides DDP, we used the following search procedures:

(1) A procedure similar to that denoted by 2) in Protein folding experiments, that generates moves one by one, allowing the engine to find, thanks to the channeling constraints, correct values for dependent variables (those belonging to `On[]` and `Clear[]`) at each time step.
(2) An enrichment of the above one, that does not generate moves that would be rejected as unfeasible by other constraints (i.e., those that try to move a block which is not clear, and those that try to put a block onto a not clear one).

Table 2
OPL solving times for benchmark instances of the HP 2D-Protein folding problem (model with the additional `Hits[]` dependent predicate), with and without search procedure ('–' means that OPL did not terminate in one hour)

Length	max. contacts	No search proc. (default strategy)	With search procedure	
			DDP	Saving %
5	0	25.12	0.09	99.64
6	0	103.18	0.10	99.90
6	2	393.59	0.14	99.96
6	0	181.98	0.10	99.95
6	1	144.64	0.17	99.88
7	0	–	0.09	100.00
7	0	–	0.16	100.00
7	2	–	0.23	>99.99
7	2	–	0.42	>99.99
8	0	–	0.12	100.00
8	1	–	0.28	>99.99
8	2	–	0.58	>99.98
8	3	–	1.85	>99.95
Total time		>29648.51	4.80	>99.98

Table 3
OPL solving times for benchmark instances of the Blocks world problem, with and without search procedures. ('–' means that OPL did not terminate in one hour)

Instance	Blocks	Min. plan length	No search proc. (default strategy)	With search procedure					
				DDP	Saving %	1.	Saving %	2.	Saving %
bw-large-a	9	12	–	23.07	>99.36	–	0.00	396.71	>88.98
bw-reversal4	4	4	0.82	0.07	91.46	0.08	90.24	0.07	91.46
bw-sussman	3	3	11.95	0.07	99.41	0.1	99.16	0.06	99.50
bw-12step	7	6	–	0.76	>99.98	–	0.00	6.57	>99.82
bw-reversal5	5	5	–	0.09	100.00	0.66	>99.98	0.10	100.00
bw-large-b	11	18	–	–	0.00	–	0.00	–	0.00
Total time			>14412.77	>3624.06	>74.86	>10800.84	>25.06	>4003.51	>72.22

The performance of DDP is often impressive in this case, leading to savings up to 99.9%. On the other hand, adding the more complex procedures 1) or 2) *does not* further improve performance. Results are shown in Table 3.

Factoring. The behavior of OPL on Factoring is slightly different than the one observed on the other problems. We made experiments with the specification shown in Appendix B.1, and added different ad-hoc built search procedures, besides DDP, in order to exploit the functional dependence of guessed predicate `carry[]` on `X[]` and `Y[]`:

(1) A procedure similar to that denoted by 1) in Protein folding experiments, that, while generating digits for the two factors from the least significant ones, *computes* carry values on the fly.
(2) A simplification of the above one, that, instead of computing carry values, leaves the engine free to search for suitable values using OPL defaults.
(3) An enrichment of procedure 1), that starts by generating equally long factors;
(4) An enrichment of 2), that starts by generating equally long factors.

The intuition behind 3) and 4) is that non-trivial instances of this problem are likely to have, as solutions, factors of comparable sizes (hence, such search procedures exploit properties related to the data, and not to the problem model).

As Table 4 shows, OPL benefits from adding DDP, saving about 19% on the average, and adding the much more complex 1) or 2) does not further improve performance. However, it is worth noting that adding 3) or 4) greatly boosts OPL, leading to savings of about 53% with respect to the specification with no search procedure. However, such procedures are strongly related to the specific problem considered and to properties of its instances, and are unlikely to be synthesized automatically.

Table 4
OPL solving times for benchmark instances of the Factoring problem, with and without search procedures

Z	No search proc. (default strategy)	With search procedure											
		DDP	Saving %	1.	Saving %	2.	Saving %	3.	Saving %	4.	Saving %	DDP + eq. long	Saving %
63,233,712,858,073	929.73	1002.18	−7.79	1049.75	−12.91	1007.73	−8.39	634.23	31.78	579.17	37.71	659.8	29.03
8,107,676,847,961	351.66	201.74	42.63	218.52	37.86	199.83	43.18	126.34	64.07	113.05	67.85	105.7	69.94
66,117,128,225,483	1111.98	1100.39	1.04	1017.26	8.52	1002.12	9.88	575.58	48.24	563.75	49.30	714.09	35.78
71,444,640,648,611	1303.51	922.78	29.21	920.48	29.38	854.58	34.44	505.5	61.22	510.01	60.87	594.89	54.36
3,457,419,019,907	193.44	128.85	33.39	135.32	30.05	127.78	33.94	78.19	59.58	67.83	64.93	34.40	82.22
37,836,417,723,859	1125.77	686.19	39.05	688.27	38.86	642.03	42.97	382.30	66.04	435.56	61.31	430.20	61.79
17,337,128,879,149	596.63	505.65	15.25	623.35	−4.48	600.82	−0.70	342.81	42.54	337.25	43.47	309.83	48.07
Total time	5612.72	4547.78	18.97	4652.95	17.10	4434.89	20.99	2644.95	52.88	2606.62	53.56	2848.91	49.24

The question that arises is how DDP performs when its behavior is enhanced by exploiting such a heuristic. Table 4 shows additional results, obtained by adding to the general DDP schema a rule that forces the engine to first generate equally long factors. Savings are much higher, and comparable to those obtained by adding 3) or 4) (about 50% on the average). This is good evidence that DDP can be considered a *good default* schema for the search procedure when dependencies among guessed predicates exist, that can be possibly enhanced by the programmer by exploiting some problem peculiarities, hard to be automated, and that *writing more complex* ad-hoc *search procedures* (in order to, e.g., compute the carries) *is unlikely to pay-off more*.

7. Discussion and future research directions

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 modeling, either because intermediate results have to be maintained in order to express some of the constraints, or because of precise choices, e.g., redundant modeling. However, dependencies can negatively affect the efficiency of the solver, since the search space can become much larger, and additional information from the programmer is currently 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 simple yet efficient search strategies (DDP) that avoid the system spending search in unfruitful branches. Several examples of constraint problems that exhibit dependencies have been presented, from bio-informatics, planning, and resource allocation, and experimental results have been discussed, showing that current systems for constraint programming greatly benefit from the addition of such search strategies.

Moreover, experimental analysis show that, almost always, spending greater effort into manually writing complex search procedures that strongly exploit the peculiarities of the particular problem to be solved, does not further improve performance. Even if, in some cases, cf., e.g., Factoring, some good intuitions (e.g., trying to generate first equally long factors) may boost solvers, their exploitation may be easily added to DDP in order to produce their results.

As claimed in Section 1, in related work we address other forms of reformulations of declarative constraint problem specifications: detection and breaking of structural symmetries [34] and the elimination of some constraints (called "safe-delay") [6], and provide semantic criteria on the specification that can be used in order to automatically perform such reasoning tasks. Despite their undecidability, in [7] we show how in practical circumstances computer tools like first-order theorem provers and finite model finders can be effectively and often efficiently used in order to mechanize the requested forms of reasoning. This, suggests, as a side-effect, a brand new application area for a technology which is undoubtedly one of the most important results achieved by Artificial Intelligence to date, and goes in the direction of building a bridge between constraint programming and deduction. In fact, although relations between these two areas have been observed since several years (cf., e.g., [2,27]), not much work has been done at the symbolic level of the specification, and available systems for constraint programming do not currently perform any kind of reasoning on the problem model.

Acknowledgements

Authors would like to thank Marco Schaerf for useful discussions and the anonymous reviewers for their comments and suggestions.

Appendix A. Proofs of results

Theorem 1. *Let* $\psi \doteq \exists \vec{\mathcal{S}}\vec{\mathcal{P}}\ \phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$ *be a problem specification with input schema* $\vec{\mathcal{R}}$. *Guessed predicates in set* $\vec{\mathcal{P}}$ *functionally depend on those in* $\vec{\mathcal{S}}$ *if and only if the following first-order formula is a tautology*:

$$\left[\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}) \wedge \phi(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}) \wedge \vec{\mathcal{S}}\vec{\mathcal{P}} \not\equiv \vec{\mathcal{S}}'\vec{\mathcal{P}}'\right] \rightarrow \vec{\mathcal{S}} \not\equiv \vec{\mathcal{S}}'. \tag{A.1}$$

Proof. *If part*. We show that if formula (A.1) is valid, then $\vec{\mathcal{P}}$ functionally depends on $\vec{\mathcal{S}}$. Actually, we prove that, if $\vec{\mathcal{P}}$ does not functionally depend on $\vec{\mathcal{S}}$, then formula (A.1) is not valid, i.e., there is an extension for $\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}$ that falsifies it. Let us assume that $\vec{\mathcal{P}}$ does not functionally depend on $\vec{\mathcal{S}}$: this means, according to Definition 1,

that there exists an instance $\vec{\mathcal{I}}$ of $\vec{\mathcal{R}}$ and two interpretations $\langle \vec{\Sigma}, \vec{\Pi} \rangle$ and $\langle \vec{\Sigma}', \vec{\Pi}' \rangle$ of predicates in $(\vec{\mathcal{S}}, \vec{\mathcal{P}})$ such that $\langle \vec{\Sigma}, \vec{\Pi} \rangle \not\equiv \langle \vec{\Sigma}', \vec{\Pi}' \rangle$, $\vec{\Sigma}, \vec{\Pi}, \vec{\mathcal{I}} \models \phi$, $\vec{\Sigma}', \vec{\Pi}', \vec{\mathcal{I}} \models \phi$ but $\vec{\Sigma} \equiv \vec{\Sigma}'$, i.e., they are models of ϕ that differ only on the extension of predicates in $\vec{\mathcal{P}}$. The interpretation $(\vec{\Sigma}, \vec{\Pi}, \vec{\Sigma}', \vec{\Pi}', \vec{\mathcal{I}})$ makes the left side of implication (A.1) true, and the right side false. Thus this interpretation is not a model of formula (A.1).

Only if part. Here we show that if $\vec{\mathcal{P}}$ functionally depends on $\vec{\mathcal{S}}$, then formula (A.1) is valid. Actually, we prove that, if formula (A.1) is not valid, i.e., there is an extension for $\vec{\mathcal{S}}, \vec{\mathcal{S}}', \vec{\mathcal{P}}, \vec{\mathcal{P}}', \vec{\mathcal{R}}$ that falsifies it, then $\vec{\mathcal{P}}$ does not functionally depend on $\vec{\mathcal{S}}$. Let us assume that an interpretation $(\vec{\Sigma}, \vec{\Pi}, \vec{\Sigma}', \vec{\Pi}', \vec{\mathcal{I}})$ of predicates $\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}$ exists that falsifies formula (A.1). This means that such an interpretation makes the left side of implication (A.1) true and the right side false. Thus, it is such that:

(1) $\vec{\Sigma}, \vec{\Pi}, \vec{\mathcal{I}} \models \phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$;
(2) $\vec{\Sigma}', \vec{\Pi}', \vec{\mathcal{I}} \models \phi(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}})$;
(3) $\vec{\Pi} \not\equiv \vec{\Pi}'$;
(4) $\vec{\Sigma} \equiv \vec{\Sigma}'$.

From points 1–4 and Definition 1, it follows that $\vec{\mathcal{P}}$ does not functionally depend on $\vec{\mathcal{S}}$, since $\langle \vec{\Sigma}, \vec{\Pi} \rangle$ and $\langle \vec{\Sigma}', \vec{\Pi}' \rangle$ are two models of ϕ that differ only for the extension of predicates in $\vec{\mathcal{P}}$. □

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

Proof. We prove the statement by reducing it to the problem of checking whether an arbitrary closed first-order formula is a contradiction.

Let $\psi \doteq \exists \vec{\mathcal{S}} \vec{\mathcal{P}}\ \phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$ be any fixed problem specification with input schema $\vec{\mathcal{R}}$, such that $\vec{\mathcal{P}}$ is functionally dependent on $\vec{\mathcal{S}}$, so, by Theorem 1:

$$\big[\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}) \wedge \phi(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}) \wedge \vec{\mathcal{S}}\vec{\mathcal{P}} \not\equiv \vec{\mathcal{S}}'\vec{\mathcal{P}}'\big] \rightarrow \vec{\mathcal{S}} \not\equiv \vec{\mathcal{S}}'$$

is a valid formula. Let $\gamma(\vec{\mathcal{R}})$ be an arbitrary closed first-order formula on the relational vocabulary $\vec{\mathcal{R}}$.

Consider the new specification $\psi' \doteq \exists \vec{\mathcal{S}} \vec{\mathcal{P}}\ \phi'(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$, where $\phi'(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}})$ is defined as $\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}) \vee \gamma(\vec{\mathcal{R}})$. From Theorem 1, $\vec{\mathcal{P}}$ functionally depends on $\vec{\mathcal{S}}$ with respect to specification ψ', if and only if

$$\big[\phi'(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}) \wedge \phi'(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}) \wedge \vec{\mathcal{S}}\vec{\mathcal{P}} \not\equiv \vec{\mathcal{S}}'\vec{\mathcal{P}}'\big] \rightarrow \vec{\mathcal{S}} \not\equiv \vec{\mathcal{S}}'$$

is a valid formula, or, equivalently,

$$\big[\big(\phi(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{R}}) \vee \gamma(\vec{\mathcal{R}})\big) \wedge \big(\phi(\vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}}) \vee \gamma(\vec{\mathcal{R}})\big) \wedge \vec{\mathcal{S}}\vec{\mathcal{P}} \not\equiv \vec{\mathcal{S}}'\vec{\mathcal{P}}'\big] \rightarrow \vec{\mathcal{S}} \not\equiv \vec{\mathcal{S}}' \tag{A.2}$$

is a valid formula. Since, by hypothesis, $\vec{\mathcal{P}}$ functionally depends on $\vec{\mathcal{S}}$ with respect to specification ψ, it follows that, if $\gamma(\vec{\mathcal{R}})$ is a contradiction, then $\vec{\mathcal{P}}$ functionally depends on $\vec{\mathcal{S}}$ with respect to ψ'.

On the other hand, let us assume that an interpretation $\vec{\mathcal{I}}$ for $\vec{\mathcal{R}}$ exists such that $\gamma(\vec{\mathcal{I}})$ is true. Consider a pair of interpretations $\vec{\Sigma}, \vec{\Pi}$ and $\vec{\Sigma}', \vec{\Pi}'$ of $(\vec{\mathcal{S}}, \vec{\mathcal{P}})$, such that $\vec{\Sigma}\vec{\Pi} \not\equiv \vec{\Sigma}'\vec{\Pi}'$ but $\vec{\Sigma} \equiv \vec{\Sigma}'$. Thus, interpretation $(\vec{\Sigma}, \vec{\Pi}, \vec{\Sigma}', \vec{\Pi}', \vec{\mathcal{I}})$ of predicates $(\vec{\mathcal{S}}, \vec{\mathcal{P}}, \vec{\mathcal{S}}', \vec{\mathcal{P}}', \vec{\mathcal{R}})$ is not a model of formula (A.2). Since, for every interpretation $\vec{\mathcal{I}}$ for $\vec{\mathcal{R}}$ (with non-empty universe) a pair of interpretations $\vec{\Sigma}, \vec{\Pi}$ and $\vec{\Sigma}', \vec{\Pi}'$ of the above kind can always be built, formula (A.2) is valid, i.e., $\vec{\mathcal{P}}$ functionally depends on $\vec{\mathcal{S}}$ with respect to ψ' if and only if $\gamma(\vec{\mathcal{R}})$ is a contradiction. Since the latter problem is not decidable, cf., e.g., [4], the former is not decidable as well. □

Appendix B. Opl code for the examples

In this appendix we show specifications of the problems described in this paper in the declarative constraint modeling language OPL [43], provided by the state-of-the-art CP system Ilog OPLSTUDIO.[6]

An OPL specification is essentially made of five parts (two of which optional):

[6] Cf. http://www.ilog.com.

- Declaration of the instance schema (name and type of parameters and relation symbols); actual values for parameters and extensions for relations (i.e., the instance) can be given in a separate file (this is denoted by a "`...`" next to their declaration);
- Declaration of the guessed predicates that encode the search space (by means of the keyword `var`); OPL allows guessed predicates to be typed, and supports functions by means of arrays;
- Optional definition of the objective function (by means of the keyword `maximize` or `minimize`);
- Specification of the constraints (in a language very similar to first-order logic, plus much syntactic sugar, like bounded integers and arithmetics over them);
- Optional definition of a search procedure, that instructs the search engine on the search strategy to follow (variable and value branching orders): in case no search procedure is given, a default strategy is applied.

It can be observed (cf. Section 2) that OPL is very similar to ESO. In particular, the keyword `var` plays exactly the same role of a second-order existential quantifier in ESO, while constraints correspond to the first-order part of an ESO specification.

After commitment to an instance (grounding), OPLSTUDIO invokes one of the two well-known commercial solvers Ilog CPLEX or Ilog SOLVER, depending on the specification being (syntactically) linear or not.

B.1. Factoring, Examples 1 and 7

```
int+ base = ...;
int+ digitsZ = ...;
range digit 0..base-1;
range positions 1..digitsZ;
digit Z[positions];

var digit X[positions];
var digit Y[positions];
var positions digitsX;
var positions digitsY;

range digitCarry 0..(base-1)*(base-1)*
                              digitsZ/base;
var digitCarry carry[1..digitsZ+1];

solve {
  digitsX + digitsY -1 <= digitsZ <=
                          digitsX + digitsY;

  // X and Y have digitsX and digitsY
  // significant digits, respectively
  X[digitsX] <> 0;
  forall (i in positions) i>digitsX => X[i] = 0;
  Y[digitsY] <> 0;
  forall (i in positions) i>digitsY => Y[i] = 0;

  // Some symmetry-breaking
  digitsX >= digitsY;

  // Smallest factor is different from 1
  not ( (digitsY = 1) & (Y[1] = 1) );

  // no carry on least significant digit
  carry[1] = 0;

  forall (i in positions)
```

```
    Z[i] = (carry[i] +
              sum (j,k in positions: j+k=i+1)
                  (X[j] * Y[k])) mod base;
  forall (i in 2..digitsZ+1)
    carry[i] = (carry[i-1] +
                sum (j,k in positions: j+k=i)
                    (X[j] * Y[k])) / base;

  // no overflow: carry for most signif. dgt is 0
  carry[digitsZ+1] = 0;
};
```

B.2. HP 2D-Protein folding, Examples 2 and 8

Model with binary inequalities

```
int+ n = ...;               // Instance schema:
                            // string length
enum Aminoacid {H,P};
range Pos [0..n-1];
range PosButLast [0..n-2];

Aminoacid seq[Pos] = ...;  // seq. of amino-acids

enum Dir {N,E,S,W};
range Coord [-(n-1)..n-1];

// Guessed predicates
var Dir Moves[PosButLast];    // Protein shape
var Coord X[Pos], Y[Pos];     // Abs coordinates
var Pos contactsNumber;

maximize contactsNumber
subject to {
 contactsNumber = (sum(t1,t2 in Pos: t1+1 < t2)
  (((seq[t1] = H) & (seq[t2] = H)) &
  ((abs(X[t1]-X[t2]) + abs(Y[t1]-Y[t2]))=1)));

 X[0] = 0; Y[0] = 0;     // Pos. of first elem.
 forall(t in Pos: t>0) { // Channeling constr's
   (Moves[t-1] = N) =>   // for X[] and Y[]
     (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]);
   };
   // Non-crossing constraint
   forall(t1, t2 in Pos : t2 > t1) {
     (X[t1] <> X[t2]) \/ (Y[t1] <> Y[t2])
   }
};
```

Model with additional guessed predicate 'Hits'

We write only the differences with respect to the previous model.

```
...
range Hit [0..n/2];  // # of hits for a cell
...
// Guessed predicates
...
var Hit Hits[Coord,Coord,Pos]; // # of times a
                               // cell is hit

maximize contactsNumber
subject to {
  contactsNumber = ...
  X[0] = 0; Y[0] = 0;     // Pos. of first elem.
  forall(t in Pos: t>0) { // Channeling constr's
   ...                    // for X[] and Y[]
  }
  // Non-crossing:
  //Initially, no cell has been hit ...
  forall (x,y in Coord: x<>0 \/ y<>0)
      Hits[x,y,0] = 0;
  Hits[0,0,0] = 1;   // ... but the origin

 // Non-crossing: Channeling constr's for Hits
 forall (t in Pos, x,y in Coord: t>0) {
   ((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]);
 };
 // Non-crossing: 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;
};
```

Search procedure of Example 11

```
search {
 X[0]=0; Y[0]=0;
 forall(t in PosButLast ordered by increasing t) {
  generate(Moves[t]);
  if Moves[t] = N then
    try X[t+1] = X[t] & Y[t+1] = Y[t]+1 endtry
  endif;
  if Moves[t] = S then
    try X[t+1] = X[t] & Y[t+1] = Y[t]-1 endtry
  endif;
  if Moves[t] = E then
    try X[t+1] = X[t]+1 & Y[t+1] = Y[t] endtry
  endif;
  if Moves[t] = W then
    try X[t+1] = X[t]-1 & Y[t+1] = Y[t] endtry
  endif;
 };
};
```

B.3. Sailco inventory, Example 9 (taken from www.ilog.com)

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

var float+ regulBoats[Periods];
var float+ extraBoats[Periods];
var float+ inv[0..nbPeriods];

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

B.4. Blocks world, Example 10

```
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];
var bool Clear[BlockOrTable, TimeWithZero];
var TimeWithZero schLen;

minimize schLen
subject to {
  forall (b in Block)       // Initial state
    On[b,0] = OnAtStart[b]; // (time 0);
  // Channeling constraints for Clear[]
  forall (b in Block, t in TimeWithZero) {
    ( ( sum(b_up in Block) (On[b_up,t]=b) ) > 0 )
    <=> (Clear[b,t] = 0);
  };
  forall (t in TimeWithZero) {
    Clear[TABLE,t] = 1;
```

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

References

[1] F. Bacchus, P. van Run, Dynamic variable ordering in CSPs, in: Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'95), Cassis, France, in: Lecture Notes in Computer Science, vol. 976, Springer, 1995, pp. 258–275.

[2] W. Bibel, Constraint satisfaction from a deductive viewpoint, Artificial Intelligence 35 (1988) 401–413.

[3] S. Bistarelli, P. Codognet, F. Rossi, An abstraction framework for soft constraints and its relationship with constraint propagation, in: Proceedings of the Fourth International Symposium on Abstraction, Reformulation and Approximation (SARA 2000), Horseshoe Bay, TX, USA, in: Lecture Notes in Computer Science, vol. 1864, Springer, 2000, pp. 71–86.

[4] E. Börger, E. Gräedel, Y. Gurevich, The Classical Decision Problem, Perspectives in Mathematical Logic, Springer, 1997.

[5] C.A. Brown, L. Finkelstein, P.W. Purdom, Backtrack searching in the presence of symmetry, in: T. Mora (Ed.), Proceedings of the Sixth International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting codes, Rome, Italy, in: Lecture Notes in Computer Science, vol. 357, Springer, 1988, pp. 99–110.

[6] M. Cadoli, T. Mancini, Automated reformulation of specifications by safe delay of constraints, Artificial Intelligence 170 (8–9) (2006) 779–801.

[7] M. Cadoli, T. Mancini, Using a theorem prover for reasoning on constraint problems, Applied Artificial Intelligence 21 (3) (2007), Special issue on "Best papers from AI*IA 2005", in press.

[8] M. Cadoli, A. Schaerf, Compiling problem specifications into SAT, Artificial Intelligence 162 (2005) 89–120.

[9] C. Calabro, R. Impagliazzo, V. Kabanets, R. Paturi, The complexity of Unique k-SAT: An isolation lemma for k-CNFs, in: Proceedings of the Eighteenth IEEE Conference on Computational Complexity (CCC 2003), Aarhus, Denmark, IEEE Computer Society Press, 2003, p. 135 ff.

[10] C.C. Chang, H.J. Keisler, Model Theory, third ed., North-Holland, 1990.

[11] B.M.W. Cheng, K.M.F. Choi, J.H.-M. Lee, J.C.K. Wu, Increasing constraint propagation by redundant modeling: an experience report, Constraints 4 (2) (1999) 167–192.

[12] J.M. Crawford, M.L. Ginsberg, E.M. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, MA, USA, Morgan Kaufmann, Los Altos, CA, 1996, pp. 148–159.

[13] P. Crescenzi, D. Goldman, C.H. Papadimitriou, A. Piccolboni, M. Yannakakis, On the complexity of protein folding, Journal of Computational Biology 5 (3) (1998) 423–466.

[14] R. Dechter, Constraint networks (survey), in: Encyclopedia of Artificial Intelligence, second ed., John Wiley & Sons, 1992, pp. 276–285.

[15] H.D. Ebbinghaus, J. Flum, Finite Model Theory, Springer, 1999.

[16] T. Ellman, Abstraction via approximate symmetry, in: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France, Morgan Kaufmann, Los Altos, CA, 1993, pp. 916–921.

[17] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in: R.M. Karp (Ed.), Complexity of Computation, American Mathematical Society, 1974, pp. 43–74.

[18] P. Flener, Towards relational modelling of combinatorial optimisation problems, in: C. Bessière (Ed.), Proceedings of the International Workshop on Modelling and Solving Problems with Constraints, in conjunction with the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, USA, 2001.
[19] R. Fourer, D.M. Gay, B.W. Kernigham, AMPL: A Modeling Language for Mathematical Programming, International Thomson Publishing, 1993.
[20] E.C. Freuder, Eliminating interchangeable values in Constraint Satisfaction Problems, in: Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91), Anaheim, CA, USA, AAAI Press/The MIT Press, 1991, pp. 227–233.
[21] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, CA, USA, 1979.
[22] E. Giunchiglia, R. Sebastiani, Applying the Davis–Putnam procedure to non-clausal formulas, in: Proceedings of the Sixth Conference of the Italian Association for Artificial Intelligence (AI*IA'99), Bologna, Italy, in: Lecture Notes in Artificial Intelligence, vol. 1792, Springer, 2000, pp. 84–94.
[23] F. Giunchiglia, T. Walsh, A theory of abstraction, Artificial Intelligence 57 (1992) 323–389.
[24] W. Hart, S. Istrail, HP benchmarks. Available at http://www.cs.sandia.gov/tech_reports/compbio/tortilla-hp-benchmarks.html (last accessed: end of 2004).
[25] T. Hnich, T. Walsh, Why Channel? Multiple viewpoints for branching heuristics, in: Proceedings of the Second International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003), Kinsale, Ireland, 2003.
[26] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), Portland, OR, USA, AAAI Press/The MIT Press, 1996, pp. 1194–1201.
[27] P.G. Kolaitis, Constraint satisfaction, databases, and logic, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, Morgan Kaufmann, Los Altos, CA, 2003, pp. 1587–1595.
[28] V. Kumar, Algorithms for constraint-satisfaction problems: A survey, AI Magazine 13 (1) (1992) 32–44.
[29] K.F. Lau, K.A. Dill, A lattice statistical mechanics model of the conformational and sequence spaces of proteins, Macromolecules 22 (1989) 3986–3997.
[30] A. Lenstra, H.W. Lenstra, Algorithms in number theory, in: J. van Leeuwen (Ed.), The Handbook of Theoretical Computer Science, vol. 1, Algorithms and Complexity, The MIT Press, 1990.
[31] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Transactions on Computational Logic 7 (3) (2006) 499–562.
[32] C.M. Li, Integrating equivalency reasoning into Davis–Putnam procedure, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000) (1), pp. 291–296.
[33] T. Mancini, Declarative constraint modelling and specification-level reasoning, PhD thesis, Università degli Studi di Roma "La Sapienza", Roma, Italy, March 2005.
[34] T. Mancini, M. Cadoli, Detecting and breaking symmetries by reasoning on problem specifications, in: Proceedings of the Sixth International Symposium on Abstraction, Reformulation and Approximation (SARA 2005), Airth Castle, Scotland, UK, in: Lecture Notes in Artificial Intelligence, vol. 3607, Springer, 2005, pp. 165–181.
[35] W. McCune, Otter 3.3 reference manual, Technical Report ANL/MCS-TM-263, Argonne National Laboratory, Mathematics and Computer Science Division, August 2003. Available at http://www-unix.mcs.anl.gov/AR/otter/.
[36] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (3–4) (1999) 241–273.
[37] N.J. Nilsson, Principles of Artificial Intelligence, Tioga Publishing Co., 1980.
[38] C.H. Papadimitriou, Computational Complexity, Addison Wesley Publishing Company, Reading, MA, 1994.
[39] P. Prosser, The dynamics of dynamic variable ordering heuristics, in: Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98), Pisa, Italy, in: Lecture Notes in Computer Science, vol. 1520, Springer, 1998, pp. 17–23.
[40] T. Pyhälä, Factoring benchmarks for SAT solvers, Technical report, Helsinki University of Technology, 2004.
[41] B.M. Smith, K. Stergiou, T. Walsh, Using auxiliary variables and implied constraints to model non-binary problems, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000) (1), pp. 182–187.
[42] L.G. Valiant, V.V. Vijay, V. Vazirani, NP is as easy as detecting unique solutions, Theoretical Computer Science 47 (3) (1986) 85–93.
[43] P. Van Hentenryck, The OPL Optimization Programming Language, The MIT Press, 1999.
[44] T. Walsh, Permutation problems and channelling constraints, in: R. Nieuwenhuis, A. Voronkov (Eds.), Proceedings of the Eighth International Conference on Logic for Programming and Automated Reasoning (LPAR 2001), Havana, Cuba, in: Lecture Notes in Computer Science, vol. 2250, Springer, 2001, pp. 377–391.
[45] D.H.D. Warren, Extract from Kluzniak and Szapowicz APIC studies in data processing, no. 24, 1974, in: Readings in Planning, Morgan Kaufmann, Los Altos, CA, 1990, pp. 140–153.